Relational Query Coprocessing on Graphics Processors

BINGSHENG HE
Microsoft Research Asia
MIAN LU
Hong Kong University of Science and Technology
KE YANG
Microsoft Corporation
RUI FANG
Highbridge Capital Management LLC, USA
NAGA K. GOVINDARAJU
Microsoft Corporation
and
QIONG LUO and PEDRO V. SANDER
Hong Kong University of Science and Technology

Graphics processors (GPUs) have recently emerged as powerful coprocessors for general purpose computation. Compared with commodity CPUs, GPUs have an order of magnitude higher computation power as well as memory bandwidth. Moreover, new-generation GPUs allow writes to random memory locations, provide efficient interprocessor communication through on-chip local memory, and support a general purpose parallel programming model. Nevertheless, many of the GPU features are specialized for graphics processing, including the massively multithreaded architecture, the Single-Instruction-Multiple-Data processing style, and the execution model of a single application at a time. Additionally, GPUs rely on a bus of limited bandwidth to transfer data to and from the CPU, do not allow dynamic memory allocation from GPU kernels, and have little hardware support for write conflicts. Therefore, a careful design and implementation is required to utilize the GPU for coprocessing database queries.

In this article, we present our design, implementation, and evaluation of an in-memory relational query coprocessing system, GDB, on the GPU. Taking advantage of the GPU hardware features, we design a set of highly optimized data-parallel primitives such as split and sort, and use these primitives to implement common relational query processing algorithms. Our algorithms

The work of Ke Yang was done while he was visiting HKUST, and the work of Bingsheng He and Rui Fang was done when they were students at HKUST.
This work was supported by grant 616808 from the Hong Kong Research Grants Council.
Authors' address: Bingsheng He; email: savenhe@microsoft.com.

DOI 10.1145/1620585.1620588 http://doi.acm.org/10.1145/1620585.1620588

utilize the high parallelism as well as the high memory bandwidth of the GPU, and use parallel computation and memory optimizations to effectively reduce memory stalls. Furthermore, we propose coprocessing techniques that take into account both the computation resources and the GPU-CPU data transfer cost so that each operator in a query can utilize suitable processors—the CPU, the GPU, or both—for an optimized overall performance. We have evaluated our GDB system on a machine with an Intel quad-core CPU and an NVIDIA GeForce 8800 GTX GPU. Our workloads include microbenchmark queries on memory-resident data as well as TPC-H queries that involve complex data types and multiple query operators on data sets larger than the GPU memory. Our results show that our GPU-based algorithms are 2–27x faster than their optimized CPU-based counterparts on in-memory data. Moreover, the performance of our coprocessing scheme is similar to, or better than, both the GPU-only and the CPU-only schemes.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing, relational databases*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Relational database, join, sort, primitive, parallel processing, graphics processors

ACM Reference Format:

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. 2009. Relational query coprocessing on graphics processors. ACM Trans. Datab. Syst. 34, 4, Article 21 (December 2009), 39 pages. DOI = 10.1145/1620585.1620588 http://doi.acm.org/10.1145/1620585.1620588

1. INTRODUCTION

Graphics processing units (GPUs) are specialized architectures traditionally designed for gaming applications. They have approximately 10x the computational power and 10x the memory bandwidth of CPUs. Moreover, new generation GPUs, such as AMD R600 and NVIDIA G80, have transformed into powerful coprocessors for general purpose computing (GPGPU). They provide general parallel processing capabilities, including support for scatter operations and interprocessor communication, as well as general purpose programming languages such as NVIDIA CUDA[1]. Recent research has shown that GPUs can be designed as accelerators for individual database operations such as sort [Govindaraju et al. 2005, 2006] and joins [He et al. 2008]. In this article, we present a query coprocessing system for in-memory relational databases on new-generation GPUs.

Similar to multicore CPUs, GPUs are commodity hardware consisting of multiple processors. However, these two types of processors differ significantly in their hardware architecture. GPUs provide a massive number of lightweight hardware threads (contexts) on over a hundred SIMD (single instruction multiple data) processors whereas current multicore CPUs typically offer a much smaller number of concurrent threads on a much smaller number of MIMD (multiple instruction multiple data) cores. Moreover, the majority of GPU transistors are devoted to computation units rather than caches, and GPU cache sizes are 10x smaller than CPU cache sizes. As a result, GPUs deliver an order of magnitude higher computational capabilities in terms of total GFLOPS (giga floating point operations per second) than current commodity multi-core CPUs.

[1]http://developer.nvidia.com/object/cuda.html.

While GPUs have been used to accelerate individual database operations, we have identified the following three challenges in developing our full-fledged GPU-based query coprocessor, GDB. The first one is how to exploit the GPU hardware features to develop a common set of relational query operators, given that parallel programming is a difficult task in general and programming the GPU for query processing is an unconventional task. To guarantee correctness and efficiency, high-level abstractions and carefully designed patterns in the software are necessary. The second challenge is the cost estimate of the GPU. The CPU-based cost model for in-memory databases [Manegold et al. 2002] may not be directly applicable to the GPU, due to the architectural differences between the CPU and the GPU. The third challenge is how to effectively coordinate the GPU and the CPU, two heterogenous processors connected with a bus of limited bandwidth for full-fledged query workloads. In particular, GPUs lack support for non-numeric data types, execute a single application at a time, and rely on the CPU for dynamic memory allocation and I/O handling.

With these challenges in mind, we develop a GPU-based query coprocessor to improve the overall performance of relational query processing. To address the programming difficulty, we propose a set of data-parallel primitives and use them as building blocks to implement a common set of relational query operators. Most of these primitives can find their functionally equivalent CPU-based counterparts in traditional databases, but our design and implementation are highly optimized for the GPU. In particular, our algorithms for these primitives take advantage of three advanced features of current GPUs: (1) the massive thread parallelism, (2) the fast inter-processor communication through local memory, and (3) the coalesced access to the GPU memory among concurrent threads.

To enable query optimization for the GPU coprocessing, we develop a cost model to estimate the total elapsed time of evaluating a query on the GPU. The model estimates the total elapsed time as the sum of three components: the time for transferring the input and the output data, and the time for GPU processing. The GPU processing time is further divided into two parts, memory stalls and computation time. The memory stalls are estimated using analytical models on the cost of fetching data from the GPU memory, and the GPU computation time using a calibration-based method. The calibration on the GPU differs from that on the CPU in the way that memory stalls are hidden—on the CPU memory stalls are hidden mainly when input data fit into CPU caches[2] whereas on the GPU they can be hidden either by on-chip local memory or by thread parallelism.

With query processing operators and cost models for the GPU, we propose GPU coprocessing techniques for evaluating a query. Each operator can be evaluated in one of the three modes, namely, on the CPU only, on the GPU only, or on both processors. The choice on the coprocessing mode is made taking costs into account in order to improve the overall performance. To allow GPU coprocessing for full-fledged SQL query workloads, we implement support for string

[2]More recent CPUs such as Sun Niagara II provide multiple contexts within a core for hiding the memory latency to some extent.

and date data types on the GPU and handle data sizes that are larger than the GPU memory (also known as the *device memory*).

We have implemented the entire GDB system, including the GPU-based operators and cost models, their CPU-based counterparts, and the coprocessing schemes. We evaluated our system on a PC with an NVIDIA GeForce 8800 GTX GPU (G80) and an Intel quad-core CPU. We used queries from the TPC-H benchmark with data sizes that either fit or exceed the device memory to evaluate the overall performance, and micro benchmarks on in-memory data for detailed studies. Our results show that: (1) our cost model for the GPU is highly accurate in estimating the total elapsed time for evaluating a query on the GPU; (2) for TPC-H queries on disk-based data, our GPU-based query processing algorithms have insignificant performance impact, since the bottleneck is disk I/O. As for in-memory data, our GPU-based query processing algorithms are 2–27x faster than their CPU-based counterparts, excluding the data transfer time between the GPU memory and the main memory. If the data transfer time is included, the speedup is 2–7x for computation-intensive operations such as joins, whereas the slowdown is 2–4x for simple operations such as selections. (3) Our coprocessing scheme assigns suitable workloads to the CPU and the GPU for improved overall performance.

The contributions of this article are as follows. First, we design and implement the first full-fledged query processor on the GPU. It includes a set of highly optimized primitives and a common set of relational operators on the GPU. Second, we develop cost models for estimating query execution time on the GPU. The query plan is optimized based on our cost models. Third, we implement support for the GPU to handle non-numeric data types and data sizes that are larger than the device memory, and propose coprocessing schemes for the GPU to improve the overall query performance.

The remainder of this article is organized as follows. In Section 2, we briefly introduce the background on the GPU architecture, and discuss related work on GPGPU techniques and architecture-aware query processing. In Section 3, we give an overview of our design and implementation of GDB. We present our cost model and coprocessing schemes in Sections 4 and 5, respectively. We experimentally evaluate our model and algorithms in Section 6. Finally, we discuss the issues of using current GPUs for query processing in Section 7, and conclude in Section 8.

2. BACKGROUND AND RELATED WORK

In this section, we introduce the GPU architecture and discuss related work on GPGPU techniques and architecture-aware query processing.

2.1 GPU

GPUs are widely available as commodity components in modern machines. They are used as coprocessors for the CPU [Ailamaki et al. 2006]. GPU programming languages include graphics APIs such as OpenGL[3] and DirectX

[3]http://www.opengl.org.

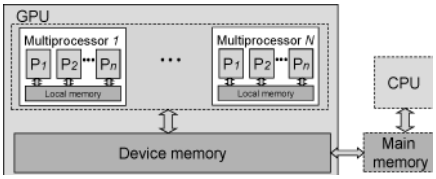


Fig. 1. The GPU architecture model.

[Blythe 2006], and GPGPU languages such as NVIDIA CUDA, AMD CTM,[4] Brook [Buck et al. 2004] and Accelerator [Tarditi et al. 2006]. The graphics APIs are used to program graphics hardware pipelines, whereas the GPGPU languages view the GPU as a general purpose processor. With the GPGPU languages, we can develop GPU programs without the knowledge of graphics hardware pipelines. For more details on the GPU and its programming techniques, we refer the reader to a recent book edited by Nguyen [2008].

The GPU architectural components that are relevant for GPGPU programming are illustrated in Figure 1. Such architecture is a common design for both AMD and NVIDIA GPUs. At a high level, the GPU consists of many SIMD multiprocessors. At any given clock cycle, each processor of a multiprocessor executes the same instruction, but operates on different data. The GPU supports thousands of concurrent threads. GPU threads have low context-switch and low creation-time compared with CPU threads. The threads on each multiprocessor are organized into *thread groups*. Threads within a thread group share the computation resources such as registers on a multiprocessor. A thread group is divided into multiple schedule units that are dynamically scheduled on the multiprocessor. Due to the SIMD execution restriction, if threads in a schedule unit must perform different tasks, such as going through branches, these tasks will be executed in sequence as opposed to in parallel. For example, branch divergence in a schedule unit causes each satisfying branch to be executed one after another on the processor. Additionally, if a thread is stalled by the memory access, the entire schedule unit will be stalled until the memory access is done.

The GPU device memory is typically in the amount of hundreds of megabytes to several gigabytes. The device memory has both a high bandwidth and a high access latency. For example, the G80 GPU has an access latency of 200 cycles and the memory bandwidth between the device memory and the multiprocessors is around 86 GB/second. Due to the SIMD property, the GPU can apply *coalesced access* to exploit the spatial locality of memory accesses among threads: when the threads in a thread group access consecutive memory addresses, these memory accesses are grouped into one. Additionally, each multiprocessor usually has a fast on-chip local memory, which is shared by all the processors in a multiprocessor. The size of this local memory is small and the access latency is low.

To develop GPU programs, developers write two kinds of code, the kernel code and the host code. The host code runs on the CPU to control the data

[4]http://ati.amd.com/products/streamprocessor/.

Table I. Notations Used in This Article

Parameter	Description
b	The memory block size of the device memory (bytes)
B_h, B_l	The memory bandwidth with and without coalesced access optimization, respectively
R, S	Outer and inner relations of the join
r, s	Tuple sizes of R and S (bytes)
$\|R\|$, $\|S\|$	Cardinalities of R and S
$\|\|R\|\|$, $\|\|S\|\|$	Total sizes of R and S (bytes)

transfer between the GPU and the main memory, and to start kernels on the GPU. The kernel code is executed in parallel on the GPU. The general flow for a computation task on the GPU consists of three steps. First, the host code allocates GPU memory for input and output data, and copies input data from the main memory to the GPU memory. Second, the host code starts the kernel on the GPU. The kernel performs the task on the GPU. Third, when the kernel execution is done, the host code copies results from the GPU memory to the main memory.

Notation. The notations used throughout this article are summarized in Table I.

2.2 GPGPU

GPGPU has been emerging in accelerating scientific, geometric, database, and imaging applications. For an overview of GPGPU techniques, we refer the reader to the recent survey by Owens et al. [2007]. We focus on the works that are most related to our study and the recent works that are not covered in the survey.

We first briefly survey the techniques that use GPUs to improve the performance of database operations. Prior to GPGPU languages, graphics APIs such as OpenGL/DirectX were utilized to accelerate database operations using the GPU. Sun et al. [2003] used the rendering and search capabilities of GPUs for spatial selection and join operations. Bandi et al. [2004] implemented GPU-based spatial operations as external procedures to a commercial DBMS. Govindaraju et al. [2004] presented novel GPU-based algorithms for relational operators including selections, aggregations, for sorting [Govindaraju et al. 2006], and for data mining operations such as computing frequencies and quantiles for data streams [Govindaraju et al. 2005]. The existing work mainly maps data structures and algorithms to textures and graphics hardware pipelines, and develops OpenGL/DirectX programs to exploit the specialized hardware features of GPUs.

GPGPU languages improve the programmability of the GPU. For example, CUDA provides C-like programming interfaces, and supports two debugging modes: CPU-based emulated execution and direct debugging on the GPU. There is much research effort in exploiting the computation power of the GPU using these languages. Sengupta et al. [2007] implemented segmented scan using the scatter. Lieberman et al. [2008] implemented a similarity join using CUDA. CUDPP [Harris et al. 2007], a CUDA library of data parallel primitives, was

released for GPGPU computing. We proposed a multipass scheme to improve the locality of the scatter and the gather operations on the GPU [He et al. 2007]. Moreover, we [He et al. 2008] proposed a set of primitives including map, split, and sort, and used the primitives to compose the four basic join algorithms including the nested-loop join with and without indexes, the sort-merge join and the hash join. In this article, we adopt the implementation of the primitives and joins [He et al. 2007, 2008], and further develop two primitives—reduce and filter—to implement a common set of relational operators such as selection and aggregation. Different from the previous studies on accelerating individual operators on the GPU, this study focuses on the design and implementation of a full-fledged query coprocessor on the GPU. In particular, we develop cost models for primitives and operators to facilitate query coprocessing on the GPU. Moreover, we propose coprocessing techniques between the CPU and the GPU for effectively utilizing the computation resources within a commodity machine.

2.3 Architecture-Aware Query Processing

Most of the traditional query processing techniques are designed for the CPU architecture [Graefe 1993]. They are optimized for a single processor, multiple processors, or a processor with multiple cores. There has also been a rich body of work on optimizing query processing performance for the memory hierarchy, especially for CPU caches.

Cache-conscious techniques [Shatdal et al. 1994; Boncz et al. 1999; Rao and Ross 1999] were proposed to reduce memory stalls by designing data structures and algorithms that better utilize CPU caches. They require the knowledge of cache parameters such as cache capacity and cache line size. In comparison, cache-oblivious algorithms [Frigo et al. 1999; He and Luo 2008] do not assume any knowledge of cache parameters but utilize the divide-and-conquer methodology to improve data locality. Our GPU-based primitives and operators are optimized for the on-chip local memory.

Parallel database systems were proposed for shared-nothing [Schneider and DeWitt 1989; Liu and Rundensteiner 2005] or shared-memory architectures [Lu et al. 1990]. These systems exploit data parallelism as well as pipelined parallelism in database workloads and utilize the aggregate computation power of multiple processors to improve performance. Hong and Stonebraker [1991] proposed a two-phase optimization strategy to reduce the search space for optimizing a parallel query execution plan. First, an optimized sequential query execution plan is created at compile time. Next, they optimize the parallelization of the sequential plan chosen from the first phase. We adopt this two-phase approach in our GPU coprocessing technique.

Recently, parallel database techniques have been investigated for multicore CPUs, including common relational operators on SMT processors [Zhou et al. 2005], data or computation sharing on chip multiprocessors [Johnson et al. 2007], and parallel adaptive aggregation in the presence of contention [Cieslewicz and Ross 2007].

In addition to query processing on CPUs, there has been recent work proposing query processing techniques on specialized architectures, including simple

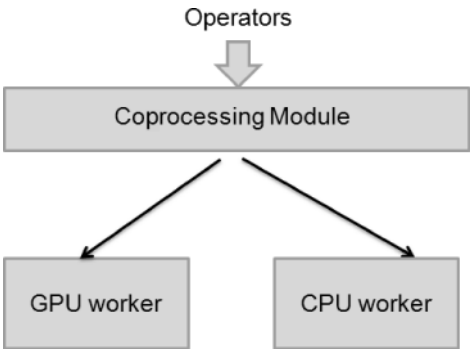


Fig. 2. The architecture of coprocessing in GDB.

query operators on network processors [Gold et al. 2005], a multi-thread hash join on the Cray MTA-2 architecture [Cieslewicz et al. 2006], as well as column databases [Héman et al. 2007], and stream joins and sorting [Gedik et al. 2007] on the cell processor. Different from previous work focusing on individual operators, this article provides a detailed description of the architectural design and implementation issues of query coprocessing on the GPU.

3. DESIGN AND IMPLEMENTATION OF GDB

In this section, we describe the architectural design and implementation of our query coprocessor, GDB. On the one hand, the GPU has 10x more computation power and memory bandwidth than the CPU. On the other hand, it relies on the CPU for task initiation, data transfer, and I/O handling. As a result, even if the execution time of a task on the GPU is much shorter than that on the CPU, the overall performance may not be improved, due to the CPU-GPU transfer cost for input data and result output.

We model the CPU and the GPU as two heterogenous processors connected with a bus of limited bandwidth. The communication between the CPU and the GPU requires explicit data transfer via the bus. For simplicity, we consider the machine consisting of one GPU and one multi-core CPU, which is a typical configuration on the current commodity machine.

In this study, we focus on optimizing a single query using the CPU and the GPU and leave multi-query optimization for future work. This simple execution model matches the GPU execution model of a single application at a time as well as the recent trend on simplifying the database engine architecture using a single-task execution model [Harizopoulos et al. 2008]. Figure 2 shows the architecture of our coprocessing scheme. We first use a Selinger-style optimizer [Selinger et al. 1979] for plan generation. Then, given a query plan, the coprocessing module is responsible for dividing the workload between the CPU and the GPU, and merging the results from the two processors if both processors are involved. It uses the CPU- and the GPU-based cost estimators to facilitate the workload partitioning. As a result, an operator in the query plan may be executed by the GPU worker, the CPU worker, or both.

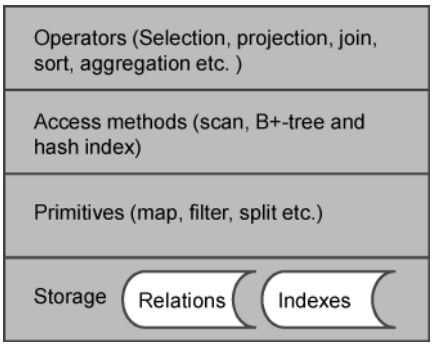


Fig. 3. The execution engine of the GPU worker in GDB.

In the remainder of this section, we focus on the GPU worker, since the design and implementation of the CPU worker is similar to that in previous work [He and Luo 2008].

3.1 GPU worker

Figure 3 shows the layered design of the execution engine of the GPU worker in GDB. We design the execution engine with four layers from bottom up, including the storage, primitives, access methods, and relational operators. Primitives are common operations on the data or indexes in the system. Our access methods and relational operators are developed based on primitives. This layered design has high flexibility. When a layer is modified, the design and implementation of the other layers requires little modification. Moreover, with the building blocks provided in the lower layer, the operations in the higher layer are easy to implement and optimize.

The system does not support online updates. Instead, it rebuilds a relation for batch updates. As in previous work [Boncz et al. 1999],[5] we focus on read-only queries. Since current GPUs do not support dynamic allocation and deallocation of the device memory within GPU kernel code, we use preallocated arrays to store relations. With the array structure, we implement our query processing algorithms on the column-based model for read-optimized performance [Stonebraker et al. 2005].

The engine supports both fixed- and variable-sized columns. The supported fixed-size columns include integer, floating point numbers, and date, and the variable-sized columns include bit and character strings. We implement a variable-sized column using two arrays, *values* and *start*. Array *values* stores the values of the column. Array *start* stores the pair of the record identifier and the start position in the *values* array for each tuple. In the remainder of the article, we focus on the algorithms for fixed-sized columns for simplicity of presentation. We have extended these algorithms for variable-sized columns in our system.

[5]http://monetdb.cwi.nl/.

Table II. Primitive Definitions

Primitive	Input	Output	Function
Map	$R_{in}[1,\ldots,n]$, a map function fcn	$R_{out}[1,\ldots,n]$	$R_{out}[i] = fcn(R_{in}[i])$
Scatter	$R_{in}[1,\ldots,n]$, $L[1,\ldots,n']$	$R_{out}[1,\ldots,n']$	$R_{out}[L[i]] = R_{in}[i], \forall i \in [1,n'] \, (n' \leq n)$
Gather	$R_{in}[1,\ldots,n]$, $L[1,\ldots,n']$	$R_{out}[1,\ldots,n']$	$R_{out}[i] = R_{in}[L[i]], \forall i \in [1,n'] \, (n' \leq n)$
Reduce	$R_{in}[1,\ldots,n]$, a reduce function $\bigodot$	$R_{out}[1]$	$R_{out}[1] = \bigodot_{i=1}^{n} R_{in}[i]$
Prefix scan	$R_{in}[1,\ldots,n]$, a binary operator $\bigoplus$	$R_{out}[1,\ldots,n]$	$R_{out}[i] = \bigoplus_{j<i}(R_{in}[j])$
Split	$R_{in}[1,\ldots,n]$, a partition function $fcn(R_{in}[i]) \in [1,\ldots,F]$, $i \in [1,n]$	$R_{out}[1,\ldots,n]$	$\{R_{out}[i], i \in [1,n]\} = \{R_{in}[i], i \in [1,n]\}$ and $fcn(R_{out}[i]) \leq fcn(R_{out}[j]), \forall i, j \in [1,n], i \leq j$
Filter	$R_{in}[1,\ldots,n]$, a filter function $fcn(R_{in}[i]) \in \{0,1\}$, $i \in [1,n]$	$R_{out}[1,\ldots,n']$	$\{R_{out}[i], i \in [1,n']\} = \{R_{in}[i] \vert fcn(R_{in}[i]) = 1, i \in [1,n]\}$
Sort	$R_{in}[1,\ldots,n]$	$R_{out}[1,\ldots,n]$	$\{R_{out}[i], i = 1,\ldots,n\} = \{R_{in}[i], i = 1,\ldots,n\}$ and $R_{out}[i] \leq R_{out}[j], \forall i, j \in [1,n], i \leq j$

3.1.1 *Primitives.* We aim at designing and implementing a complete set of data parallel primitives on the GPU for query processing. The definitions of these primitives are shown in Table II.

Our primitive-based approach allows us to use different primitive implementations. For example, we can adopt the implementation from previous studies [He et al. 2007, 2008], or from CUDPP [Sengupta et al. 2007; Harris et al. 2007]. Since the implementation in previous studies has been used for implementing joins and their efficiency has been carefully studied [He et al. 2007, 2008], we mainly adopt their implementation of primitives and joins. Specifically, we adopt scatter, gather, and hash search [He et al. 2007], map, split, sort, and GPU-based CSS-Tree indexing and joins [He et al. 2008], and the prefix scan from CUDPP [Sengupta et al. 2007; Harris et al. 2007]. To make our presentation self-contained, we briefly review the implementation of these primitives and query-processing algorithms.

The primitives have the following features:

(1) They exploit the GPU hardware features, especially the high thread parallelism and the fast local memory to reduce the memory stalls.
(2) They are scalable to hundreds of processors, because all primitives are designed without locks and the synchronization cost is low by using the fast local memory.

—*Map*. A *map* operation is defined the same as a mapping function in Lisp. Given an array of data tuples and a function, a map applies the function to every tuple. An example of the map operator is to evaluate a predicate on

a relation. We adopt the implementation [He et al. 2008] for the map. The map uses multiple thread groups to scan the relation. Each thread group is responsible for a segment of the relation. The access pattern of the threads in each thread group is designed to exploit the coalesced access feature on the GPU.

—*Scatter and gather*. A *scatter* operation performs indexed writes to a relation, for example, hashing. The location array, L, defines a distinct write location for each input tuple. A *gather* primitive performs indexed reads from a relation. It can be used, for instance, when fetching a tuple given a record ID, and probing hash tables. The location array, L, defines the read location for each output tuple. When locations are sequential, the scatter and the gather are the same as the map operation. When locations are random, the scatter and the gather have bad memory locality due to the random accesses. We implemented the scatter and the gather using the multipass optimization scheme [He et al. 2007]. In each pass, the scatter writes to a certain region in the output array; the gather reads the data from a certain region in the input array. The multipass optimization improves the temporal locality of the scatter and the gather.

—*Prefix scan*. A *prefix scan* operation applies a binary operator to the input relation. An example of prefix scan is the prefix sum, which is an important operation in parallel databases [Blelloch 1990]: given an input relation (or array) R_{in}, the value of each output array tuple $R_{out}[i]$ $(2 \leq i \leq |R_{in}|)$ is obtained from the sum of $R_{in}[1], \ldots,$ and $R_{in}[i-1]$ $(R_{out}[1] = 0)$. We adopt the implementation from CUDPP [Sengupta et al. 2007; Harris et al. 2007], which is highly parallel and efficient on the GPU.

—*Split*. A *split* primitive divides a relation into a number of disjoint partitions according to a given partitioning function. The result partitions are stored in the output relation. Splits are used in hash partitioning or range partitioning.

We adopt the lock-free implementation for the split [He et al. 2008]. It uses histograms to compute the write location for each tuple (stored in the array L) and scatters R_{in} to R_{out} according to the array L. Since each thread knows its target position to write, the write conflicts among threads are avoided. Without locks, the synchronization of all threads is implicitly performed as the creation and termination of the kernels in the implementation.

In the algorithm, each thread is responsible for a portion of the relation. Given the total number of threads, $\#thread$, and the split fanout, F, the split primitive works in the following five steps. First, each thread constructs its local histogram from R_{in}. The thread stores the histogram in the array $tHist[t][1, \ldots, F]$. Second, each thread writes its histogram to L. For thread t, we have $L[(p-1) * \#thread + t] = tHist[t][p]$ $(1 \leq p \leq F)$. Third, the algorithm performs a prefix sum on L. The result is stored in L. Fourth, each thread updates its local offset array. The thread t has a local offset array $tOffset[t][1, \ldots, F]$ so that $tOffset[t][p] = L[(p-1) * \#thread + t]$ $(1 \leq p \leq F)$. Finally, each thread scatters its tuples to R_{out} based on its local offset array. The five steps are implemented using our other primitives. The first step is implemented using a map primitive; the third a prefix scan; the fourth a

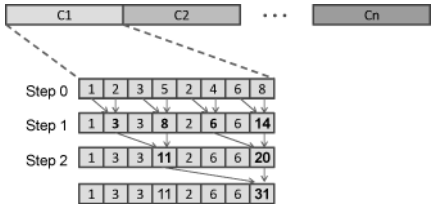


Fig. 4. One pass of the reduce primitive.

gather; and the other two a scatter. Since the histogram and the offset array are accessed frequently, we store them in the local memory.

—*Sort*. The *sort* operation transforms an array of unordered data into an array of ordered data. It is used in a number of operators such as order-by and join operators. We adopt the quick sort on the GPU [He et al. 2008]. The quick sort is implemented using the split primitive. The quick sort has two steps. First, given a set of pivots, the algorithm uses the split primitive to divide the relation into multiple chunks. The pivots are chosen randomly. The split process goes recursively until each chunk is smaller than the local memory size. In the second step, multiple chunks are sorted in parallel, and each chunk is sorted using the bitonic sort.

To implement a full set of relational operators, we implement two more primitives: reduce and filter. We optimize the primitives and the operators on the GPU so that they exploit the thread parallelism of the GPU, and take advantage of coalesced access for spatial locality and local memory optimization for temporal locality.

—*Reduce*. A *reduce* operation computes a value based on the input relation. For example, a reduction can be used to compute the sum of all the key values in a relation. We implement the reduce primitive as a multipass algorithm by utilizing local memory optimization. Given the relation R_{in}, the reduce primitive works in $\log_{\frac{M}{r}} |R_{in}|$ passes, where M is the local memory size per multiprocessor (bytes). In each pass, we first divide the input data into multiple chunks and evaluate them in parallel. Each chunk has the size of the local memory. The reduce operation on each chunk is performed entirely in the local memory, thus improving temporal locality. The reduce operation of each chunk has $\log_2 C$ steps, where C is the number of tuples that can fit into the local memory. In step i ($0 \leq i < \log_2 C$), thread j computes the partial reduction of $R_{in}[j \times 2^i]$ and $R_{in}[(j+1) \times 2^i]$. The computation of each pass is shown in Figure 4. The reduction result of each chunk forms the input array for the next pass.

—*Filter*. A *filter* primitive selects a subset of elements from a relation, and discards the rest. It can be used in the selection operator. We use the map, the prefix scan, and the scatter primitives to implement the filter. The filter works in three steps. First, we use the map primitive to process the input array and obtain the corresponding 0-1 result array. We use the filter function as the map function. The result is stored in the array, $flag[1, \ldots, n]$. Second, we compute a prefix sum on *flag*, and store the prefix sum into another array

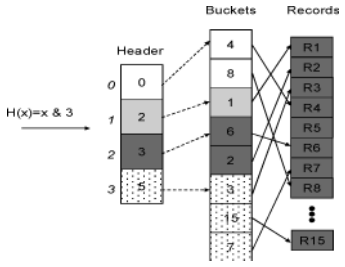


Fig. 5. An example of hash indexes.

ps. The sum of *flag* is the number of tuples generated by the filter. Third, we scatter the result to the output array according to the *flag* and *ps* arrays. For the ith tuple of the input array, if $flag[i] = 1$, $ps[i]$ is the position for outputting the tuple to the output array.

3.1.2 *Access methods.* The engine supports three common access methods, including the table scan, the B+- tree, and the hash index.

—*Table scan*. The table scan is implemented using the map primitive. If the relation is sorted, a binary search is performed on the relation according to the sort key.

—*B+-trees*. We adopt the GPU-based CSS-Tree [He et al. 2008] as our tree index. The CSS-Tree stores the entire tree in an array. This structure fits well into our storage model on the GPU. Additionally, the tree traversal on the CSS-Tree is performed via address arithmetic. This effectively trades off more computation for less memory access, which makes it a suitable index structure to utilize the GPU's computational power.

The tree search consists of two major steps, searching for the first occurrence of matching tuples in the indexed relation, and then accessing the indexed relation for matching results. Multiple keys are processed in parallel on the tree. The search starts at the root node and steps one level down the tree in each iteration until it reaches the data nodes on the bottom.

—*Hash indexes*. The GPU-based hash table consists of two arrays, namely headers and buckets [He et al. 2007]. Each element in the header array maintains the start position to its corresponding bucket. Each bucket stores the key values of the records that have the same hash value, and the pointers to the records. An example of the hash table is illustrated in Figure 5.

The hash search uses a thread for one search key. The hash search is performed in four steps. First, for each search key, the algorithm uses the map primitive to compute its corresponding bucket ID and the gather primitive to fetch the $(start, end)$ pair indicating the start and end locations for the bucket. Second, it scans the bucket and determines the number of matching results. Third, based on the number of results for each key, the algorithm computes a prefix sum on these numbers to determine the start location at which the results of each search key are written, and then outputs the record IDs of the matching tuples to an array, L. Fourth, a gather is performed according to L to fetch the actual result records.

3.1.3 *Relational Operators.* The engine supports the following common query processing operators.

—*Selection*. In the absence of indexes, the selection is directly implemented using the filter primitive. The predicate evaluation is the filter function. In the presence of indexes, if the selectivity is high, a filter primitive on the relation is performed. Otherwise, the B+-tree index or the hash index can be used.

—*Projection*. Given an array of record IDs, the projection operator extracts the tuples from the relation. We implement the projection using the gather primitive. The read locations are given by the record IDs. If duplicate elimination is required, we use sorting to eliminate the duplicates for the projection.

—*Ordering*. We implement order-by operator using the sort primitive.

—*Grouping and aggregation*. We use the sort primitive to perform grouping and the reduce primitive for aggregation.

—*Joins*. We adopt the four joins [He et al. 2008], including the nested-loop join with or without indexes, the sort-merge join, and the hash join.

To avoid the conflicts between concurrent writes in the result output, all joins use a three-step output scheme [He et al. 2008]. First, each thread counts the number of join results for the partitioned join it is responsible for. Second, we compute a prefix sum on the counters to get an array of write locations, each of which is the start location in the device memory for the corresponding thread to write. Third, the host code allocates a memory area of the exact size of the join result and each thread outputs the join results to the device memory according to its start write location. This three-phase scheme is lock-free, and does not require the hardware support of atomic functions.

For simplicity, we consider the join on two relations R and S, and assume $\|R\| \leq \|S\|$.

—*Non-indexed NLJs (NINLJ)*. The nested-loop join is blocked nested-loops that can be naturally mapped to our GPU model. Each thread group computes the join on a portion of R and S, denoted as R' and S', respectively. The size of S' is set to the local memory size. Within a thread group, each thread processes the join on one tuple from R' and all tuples from S'.

—*Indexed NLJs (INLJ)*. The process of INLJ is the same as the tree search. Each thread is responsible for searching one key against the tree index.

—*Sort-merge joins (SMJ)*. Similar to the traditional sort-merge join, we first sort the two relations and then perform a merge step on these two sorted relations. The merge is performed in three steps. First, we divide the smaller relation, S, to be Q chunks ($Q = \frac{\|S\|}{M}$). The size of each chunk (except the last chunk) is M, so that each chunk fits into the local memory. Second, we use the key values of the first and the last tuples of each chunk in S to identify the start and the end positions of its matching chunks in R. Third, we merge each pair of the chunk in S and its matching chunk in R in parallel.

—*Hash joins (HJ)*. The GPU-based HJ is a parallel version of the radix hash join [Boncz et al. 1999]. The HJ has two phases. First, both R and S are split

into the same number of partitions using radix bits so that most S partitions fit into the local memory. The join on R and S is decomposed into multiple small joins on an R partition and its corresponding S partition. In the second phase, multiple small joins are evaluated in parallel using NINLJ.

4. COST ESTIMATION FOR THE GPU

We develop a cost model for estimating the elapsed time of evaluating a query on the GPU. The total elapsed time of evaluating a query on the GPU may include the data transfer time between the device memory and the main memory, depending on whether the input and the output data are stored in the device memory. We estimate the total elapsed time of evaluating a query on the GPU as $T_{overall}$ in Equation 1.

$$T_{overall} = T_{mm\_dm}(I) + T_{GPU} + T_{dm\_mm}(O). \quad (1)$$

The total cost is the sum of three components:

(1) $T_{mm\_dm}(I)$ is the time for copying the input data from the main memory to the device memory. I denotes the set of input data.
(2) T_{GPU} is the time of evaluating the query, given the input data already in the device memory. The output data are stored in the device memory.
(3) $T_{dm\_mm}(O)$ is the time for copying the query result from the device memory to the main memory. O denotes the set of output data.

If the input and the output data are in the device memory, we exclude the time for the data transfer between the device memory and the main memory.

Similar to a previous CPU-based cost model [Manegold et al. 2002], our model estimates the elapsed time on the GPU in two parts, namely the memory access time and the computation time. Equation 2 gives the estimation on T_{GPU}, where T_{Mem} and $T_{computation}$ denote the memory access time and the computation time, respectively.

$$T_{GPU} = T_{Mem} + T_{computation}. \quad (2)$$

It is challenging to develop an accurate cost model for the time components on the GPU, since the GPU is a massively parallel processor. Moreover, unlike CPU vendors, GPU vendors do not expose much detail about the GPU architecture. Due to the architectural difference between the two processors, the CPU-based cost models on T_{Mem} and $T_{computation}$ [Manegold et al. 2002] are not applicable to the GPU. To address these challenges, we categorize the time components in our cost model into two kinds. The estimation of the first kind, including T_{mm_dm}, T_{dm_mm}, and T_{Mem} can be derived using an analytical method given a small set of hardware parameters. For the second kind, $T_{computation}$, we treat the GPU as a black box and use a calibration-based approach to estimate the cost.

4.1 Estimating T_{mm_dm} and T_{dm_mm}

We first develop the cost model for the data transfer between the main memory and the device memory. We use the same model for T_{mm_dm} and T_{dm_mm}, and

assume the bandwidth from the main memory to the device memory is the same as that from the device memory to the main memory. We denote this bandwidth to be $Band$ bytes/sec.

We model the cost of a data transfer between the main memory and the device memory as the sum of two parts, the initialization cost of invoking the transfer and the time for transferring the data. Given the size of the data chunk to be transferred, x bytes, the cost of transferring the data chunk from the main memory to the device memory is given by Equation 3. This formula applies to estimating T_{dm_mm} as well.

$$T_{mm\_dm}(x) = T_0 + \frac{x}{Band}. \tag{3}$$

In the model, T_0 is the initialization cost of invoking the transfer (in seconds). After the initialization, the data chunk is transferred at $Band$ bytes/sec, fully utilizing the memory bandwidth.

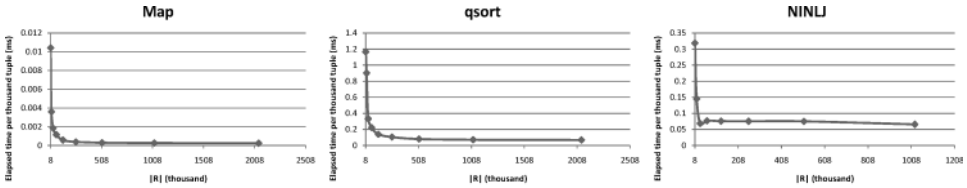
4.2 Estimating $T_{computation}$

Due to the highly parallel architecture of the GPU, we propose a calibration-based method using micro benchmarks to estimate the computation cost for the GPU. This calibrated computation time includes the access time for the local memory in addition to the pure computation time.

Specifically, we view the GPU as a black box and measure the *unit cost* of our query processing algorithms. The unit cost of an algorithm is defined to be the total cost divided by the work complexity, $O(N_1, \ldots, N_q)$, with a constant factor of one (q is the number of relations involved in the algorithm, and N_i is the total number of tuples in the ith relation). For example, the unit cost of a map is the total time of a map divided by $|R|$, and the unit cost of an NINLJ is the total time of the NINLJ divided by $|R| \times |S|$.

The key issue of calibrating the computation time is to determine the suitable input sizes so that the memory stalls are zero. The CPU-based calibration [Manegold et al. 2002] assumes the memory stall is zero only when the input fits into the cache. This assumption holds on the CPU where thread parallelism is relatively low. In contrast, memory stalls can be hidden by the on-chip local memory as well as by the massive thread parallelism on the GPU. Thus, on the GPU we need to find the input sizes with which the memory stalls can be fully hidden by both factors. In general, due to the massive thread parallelism, the input sizes for calibrating the GPU costs are larger than those obtained from considering only the local memory size. As such, adopting the traditional approach may overestimate the computation cost for the GPU.

We experimentally measured the unit cost of our query processing algorithms with the data size varied. The experimental setup is described in Section 6. Figure 6 shows the unit cost of the map, the quick sort, and the NINLJ. The map has the lowest computation cost, the NINLJ the highest, and the quick sort is in the middle. We observe similar performance trends for other algorithms with the data size varied. The unit cost is very high for small data sets (e.g., smaller than eight thousand tuples), and decreases as the data set size increases. This is because the memory stalls are not fully hidden when the data size is small,



(a) Map (b) quick sort (c) NINLJ ($|R| = |S|$)

Fig. 6. The measured unit cost. The unit cost is very high for small data sets, and decreases as the data set size increases.

and more memory stalls are hidden as the data size increases. The unit cost reaches the minimum at a *balanced* point. As the data size increases from the balanced point, the unit cost may increase until it becomes almost constant. The balanced point varies with different algorithms. The balanced points for the map, quick sort, and the NINLJ are 2 MB, 8 MB, and 256 KB, respectively.

Based on the performance trend, we use micro benchmarks to find the balanced point for each algorithm. Given the balanced point $(n_0, \ldots, n_q)$ for the sizes of the q inputs to the algorithm, we estimate the unit cost of the algorithm to be μ, where T^0_{GPU} is the estimated computation cost.

$$\mu = \frac{T^0_{GPU}}{O(n_0, \ldots, n_q)}. \tag{4}$$

We estimate T^0_{GPU} by subtracting the estimated memory cost and the memory transfer cost from the calibrated total cost.

Given the unit cost in the computation of an algorithm, μ, we estimate the computation time of the algorithm in Equation 5.

$$T_{computation} = \mu \cdot O(N_1, \ldots, N_q). \tag{5}$$

4.3 Estimating T_{Mem}

We first derive the analytical cost model for the primitives and then for the query processing algorithms. The basic idea of estimating the memory access time T_{Mem} is to estimate the total size of data accesses to the device memory. We compute the time to be the ratio of the data size and the bandwidth of the device memory. We exclude the accessing cost of the local memory from T_{Mem}, since the local memory cost is already included in $T_{computation}$.

When deriving the cost model for the memory access time, we distinguish the coalesced and noncoalesced access patterns, because the coalesced access pattern has a higher memory bandwidth than the noncoalesced one.

Equation 6 gives the cost of the map primitive, C_{map}, where B_h is the memory bandwidth of coalesced accesses to the device memory.

$$C_{map} = \frac{\|R_{in}\| + \|R_{out}\|}{B_h}. \tag{6}$$

Since the locality of data accesses is important for the scatter and the gather [He et al. 2007], we estimate their cost in two aspects in Equations 7 and 8, respectively. When the locations are sequential, accesses to the input array, the

location array and the output array are coalesced. The costs of both the scatter and the gather are estimated to be $\frac{\|R_{in}\|+\|R_{out}\|+\|L\|}{B_h}$. When the locations are random, accesses to the output array in the scatter have low locality. Each write results in an access to the device memory, and the accesses are not coalesced. Thus, we estimate the scatter cost to be $\frac{\|R_{in}\|+\|L\|}{B_h} + \frac{|R_{out}|\cdot b}{B_l}$, where b is the block size of the device memory (in bytes). Similarly, we obtain the cost of the gather to be $\frac{\|R_{out}\|+\|L\|}{B_h} + \frac{|R_{in}|\cdot b}{B_l}$. When there is no knowledge about whether the locations are random or sequential, we assume the access locations are random for worst-case estimation.

$$C_{scatter} = \begin{cases} \frac{\|R_{in}\|+\|R_{out}\|+\|L\|}{B_h}, & L \text{ is sequential} \\ \frac{\|R_{in}\|+\|L\|}{B_h} + \frac{|R_{out}|\cdot b}{B_l}, & \text{otherwise.} \end{cases} \tag{7}$$

$$C_{gather} = \begin{cases} \frac{\|R_{in}\|+\|R_{out}\|+\|L\|}{B_h}, & L \text{ is sequential} \\ \frac{\|R_{out}\|+\|L\|}{B_h} + \frac{|R_{in}|\cdot b}{B_l}, & \text{otherwise.} \end{cases} \tag{8}$$

The reduce primitive has $\log_{\frac{M}{r}} |R_{in}|$ passes. In pass i, the reduce primitive reads the array sized $\frac{\|R\|}{M^i}$ bytes, and writes the array sized $\frac{\|R\|}{M^{i+1}}$ bytes, which subsequently becomes the input array for the next pass. The reads are coalesced, whereas the writes are not. Equation 9 gives the cost estimation of the reduce primitive.

$$C_{reduce} = \frac{\|R_{in}\|}{M-1} \cdot \left(\frac{M}{B_h} + \frac{1}{B_l}\right). \tag{9}$$

Since the prefix scan consists of a reduce and a down-sweep, and the down-sweep stage is similar to the reduce stage, we estimate the total cost of the prefix scan as twice of the cost of the reduce primitive (Eq. 10).

$$C_{pscan} = 2 \cdot C_{reduce}. \tag{10}$$

We estimate the cost of the split as the sum of the five steps' costs. The first step is a sequential read on the input array with a cost of $\frac{\|R_{in}\|}{B_h}$. Step 2 is a coalesced write for the histogram array. Suppose the total size of histograms is H bytes, the cost of Step 2 is $\frac{H}{B_h}$. In our experiments, each element in the histogram is an integer of four bytes. Given the split fanout, F, $H = 4 \cdot \#thread \cdot F$ in our experiment. Step 3 is a prefix sum on the histogram array of $\#thread \cdot F$ integers. The cost is C_{pscan}. Step 4 loads the array storing the prefix sum into the local memory, resulting in a cost of $\frac{4 \cdot \#thread \cdot F}{B_h}$. Step 5 is a random scatter, and the cost is $C_{scatter}$. Thus, the total cost of the split primitive is given by C_{split} in Equation 11.

$$C_{split} = \frac{\|R_{in}\|}{B_h} + \frac{4 \cdot \#thread \cdot F}{B_h} + C_{pscan} + \frac{4 \cdot \#thread \cdot F}{B_h} + C_{scatter}. \tag{11}$$

We estimate the cost of the filter as the sum of a map, a prefix sum and a sequential scatter, as shown in Equation 12.

$$C_{filter} = C_{map} + C_{pscan} + C_{scatter}. \tag{12}$$

The quick sort consists of $\log_F \frac{\|R_{in}\|}{M}$ passes of split. For simplicity, we estimate the cost of each pass to be the same. Additionally, sorting the chunks generated by the split requires fetching the entire relation into the local memory and storing the sorted relation into the device memory. The total cost of the quick sort is given as C_{qsort} in Equation 13.

$$C_{qsort} = \log_F \frac{\|R_{in}\|}{M} \cdot C_{split} + 2\frac{\|R_{in}\|}{B_h}. \tag{13}$$

Given the fanout of the tree index, f, the height of the tree index is $\log_f |R|$. Denote the number of levels of the tree index that can fit into the local memory to be $l_{max} = \log_f(\frac{M}{z} \cdot (f-1)+1)$, where z is the tree node size in bytes. The average number of levels of the tree index accessed by each search is $(\log_f |R| - l_{max})$. Since accesses to the tree index are not coalesced, the average cost of a search over the tree index is C_{tree} in Equation 14.

$$C_{tree} = \frac{z \cdot \left(\log_f |R| - \log_f \left(\frac{M}{z} \cdot (f-1)+1\right)\right)}{B_l}. \tag{14}$$

Suppose the number of buckets of the hash table is $\#bucket$. Denote the entry size of the header array and the bucket array to be h and e bytes, respectively. The average number of records in a bucket is $\frac{|R|}{\#bucket}$. Due to the random access pattern of the hash search, the accesses in hash search, except those to the header array, are not coalesced. Thus, the average cost of a search over the hash table is C_{hash} in Equation 15.

$$C_{hash} = \frac{h}{B_h} + \frac{\left(b \cdot \lceil \frac{2e}{b} \rceil + \lceil \frac{r}{b} \rceil \cdot \frac{|R|}{\#bucket}\right)}{B_l}. \tag{15}$$

Since all our joins use the three-step output scheme, we first estimate the cost of the output scheme. Since the matching cost depends on the join algorithm, we estimate the cost of the output scheme excluding the matching cost. Suppose the total number of threads in the output scheme is T, the first step writes T integers and the third step reads T integers. These accesses are coalesced. The total cost is $2 \cdot \frac{4 \cdot T}{B_h} = \frac{8 \cdot T}{B_h}$. The cost of the second step is C_{pscan}. Suppose the total size of join results is O bytes. Since the writes in the output are not coalesced, the cost is $\frac{O}{B_l}$.

$$C_{output} = \frac{8 \cdot T}{B_h} + \frac{O}{B_l}. \tag{16}$$

Since each thread group accesses R' and S' only once from the device memory, the total volume of data transfer between the GPU and the device memory is $\frac{|R| \cdot |S|}{|R'| \cdot |S'|}(\|R'\| + \|S'\|) = \frac{|R| \cdot |S|}{T \cdot M}(T \cdot r \cdot s + M \cdot s)$. The counting step in the three-step output scheme doubles the cost [He et al. 2008]. The cost of performing NINLJ on R and S is given by C_{ninlj} in Equation 17.

$$C_{ninlj} = 2 \cdot \frac{|R| \cdot |S|}{T \cdot M \cdot B_h}(T \cdot r \cdot s + M \cdot s) + C_{output}. \tag{17}$$

We estimate the cost of INLJ, SMJ, and HJ in Equations 18, 19 and 20, respectively. We estimate the cost of the matching step to be $\frac{\|R\|+\|S\|}{B_h}$ for the

three joins. Additionally, the counting step in the three-phase output scheme doubles the matching cost.

$$C_{inlj} = |S| \cdot C_{tree} + 2 \cdot \frac{\|R\| + \|S\|}{B_h} + C_{output} \tag{18}$$

$$C_{smj} = C^R_{sort} + C^S_{sort} + 2 \cdot \frac{\|R\| + \|S\|}{B_h} + C_{output} \tag{19}$$

$$C_{hj} = C^R_{split} + C^S_{split} + 2 \cdot \frac{\|R\| + \|S\|}{B_h} + C_{output}. \tag{20}$$

5. GPU COPROCESSING

Assigning an operator to suitable processor(s) is important for the effective utilization of computation resources. Since the GPU has a much higher computation capability and memory bandwidth than the CPU, GPU processing is potentially more efficient than CPU processing when the input data resides in the GPU device memory. However, if we include the cost of data transfer between the main memory and the GPU memory, GPU processing can be slower than CPU processing, for example, on some simple operators such as selections. Finally, in the case of involving both the CPU and the GPU for an operator, the data partitioning scheme between the two processors needs to be determined. Therefore, we propose a coprocessing scheme that assigns each operator in a query plan to suitable processors.

Given a query plan, the coprocessing scheme determines how each operator in a query plan is evaluated using a cost-based approach. There are three modes of evaluating an operator: on the CPU only, on the GPU only, and using both processors, denoted as EXEC_CPU, EXEC_GPU, and EXEC_COP, respectively. The intuition for coprocessing is that EXEC_CPU is suitable for simple operators such as selections, whereas EXEC_GPU and EXEC_COP are suitable for complex operators such as joins. Evaluating an operator in EXEC_COP and EXEC_GPU may require partitioning the input data. In EXEC_GPU, when the data is larger than the GPU device memory, the input needs to be partitioned in order to fit into the GPU device memory. In EXEC_COP, the workload is divided between the CPU and the GPU.

When generating a coprocessing plan, we adopt the two-phase approach [Hong and Stonebraker 1991]: first we generate a query plan using a Selinger-style optimizer [Selinger et al. 1979] for the CPU, and then for each operator we determine whether to assign it to the CPU, to the GPU, or partition it to both the CPU and the GPU. For each operator, the cost model takes into consideration whether the input data is in the main memory or already in the device memory. If a query plan consists of a small number of nodes (the threshold is ten in our experiment), we consider all the combinations, and pick the plan with the minimum cost. Otherwise, we decompose the query plan into multiple subqueries, each of which has a small number of nodes less than the threshold. We determine the optimal query plans for the subqueries in their post order, and obtain the final plan as the combined plan from all the subqueries.

This is a tradeoff between the optimality of the query plan and the efficiency of generating the optimal plan.

As described in the coprocessing algorithm, partitioning is required in evaluating an operator in EXEC_GPU or EXEC_COP. Following the divide-and-conquer methodology, the evaluation has three steps:

(1) *Partitioning*. We divide the operator into multiple independent suboperators. The memory footprint of the suboperator executing on the GPU is bounded by the device memory. Moreover, we consider the performance difference of evaluating the operator on the CPU and on the GPU to determine the granularity of the suboperators.
(2) *Suboperator execution*. We execute the corresponding suboperators on the CPU or the GPU.
(3) *Merging*. We merge the results generated by the suboperators into the final result on the CPU.

Partitioning and merging are well studied techniques in parallel databases [DeWitt and Gray 1992]. We now briefly describe our implementation on the sort and the join. Since the EXEC_GPU scheme is similar to the EXEC_COP scheme, we present the implementation for EXEC_COP scheme only.

We design the sort in EXEC_COP as a merge sort. Suppose the maximum number of tuples that can be sorted on the GPU is c_0. Let p be the ratio of the estimated elapsed time of sorting c_0 tuples on the GPU over that on the CPU. We set the task sizes for the CPU and the GPU to be $(c_0 \times p)$ and c_0, respectively. When either processor becomes available, we assign a task with the corresponding size to that processor. Finally, we merge the sorted chunks into a sorted relation with the CPU-based merge sort. The merge sort is efficient on the CPU, because it requires only a single sequential scan on the two sorted input relations.

The EXEC_COP scheme for the NLJs with and without indexes both perform a horizontal partitioning on the outer relation. The partition sizes are determined similar to that for the sort. Thus, the NLJs are divided into multiple subjoins. In each subjoin, the NINLJ and the INLJ use a chunk of the outer relation to join the inner relation via nested loops or the tree index. Finally, we combine the results generated by the subjoins into the final result on the CPU.

The EXEC_COP scheme in the SMJ sorts the two input relations with EXEC_COP, and the merge phase is performed on the CPU.

The EXEC_COP scheme of HJ involves a hash partitioning on both R and S into a small number of partitions. We choose both processors to handle the partitioning on one relation for parallel processing, and then merge the generated partitions on the CPU. After the hash partitioning, the hash join is decomposed into multiple subjoins on R partitions and their corresponding S partitions. We dynamically assign the subjoins to the processors. The task assignment is based on the performance ratio of evaluating the hash join on the CPU only and on the GPU only. If the GPU-based hash join is faster than its CPU-based counterpart, we assign the subjoin with the largest estimation cost to the GPU and the subjoin with the smallest estimation cost to the CPU.

Table III. Hardware Configuration

	GPU	CPU(Quad-core)
Processors	1350MHz×8×16	2.4 GHz × 4
Data cache (local memory)	16KB ×16	L1: 32KB×4, L2: 4096KB×2
Cache latency (cycle)	2	L1: 2 , L2: 8
DRAM (MB)	768	2048
DRAM latency (cycle)	200	300
Bus width (bit)	384	64
Memory clock (GHz)	1.8	1.3

A final note is that, even though coprocessing involving both the CPU and the GPU for a query operator utilizes the aggregate power of both processors, the overall performance improvement may be insignificant due to two factors. The first factor is the relative performance of the GPU- and CPU-based algorithms. If a GPU-based algorithm is much faster than its CPU-based counterpart, little load will be given to the CPU, and the performance gain of partitioning will be low. The second factor is the runtime overhead of partitioning, including dividing the workload, merging the intermediate and final results, and the synchronization between the CPU and the GPU workers.

6. EXPERIMENTAL RESULTS

We study the overall performance of GDB using TPC-H benchmark queries on disk-resident data in comparison with a commercial database engine. We then perform detailed studies on our cost model and individual algorithms using micro benchmarks on memory-resident data.

6.1 Experimental Setup

We have implemented and tested our algorithms on a PC with an NVIDIA GeForce 8800 GTX GPU and a recently-released Intel Core2 Duo processor. The hardware configuration of the PC is shown in Table III. The GPU uses a PCI-EXPRESS bus to transfer data between the main memory and the device memory with a theoretical bandwidth of 4 GB/s.

We compute the theoretical bandwidth to be the bus width multiplied by the memory clock rate. The GPU and CPU have theoretical bandwidths of 86.4 GB/s and 10.4 GB/s, respectively. Based on our measurements, the G80 achieves a memory bandwidth of around 69.2 GB/s whereas the quad-core CPU has 5.6 GB/s.

We use both the TPC-H benchmark and micro benchmarks to evaluate the performance of GDB. The experiments using the TPC-H benchmark are to compare the overall performance, and those on micro benchmarks to assess the effectiveness of individual techniques in GDB.

We choose a subset of queries from the TPC-H benchmark such that the queries cover different data types such as date and string, and complex operators such as joins and sorting. Specifically, we choose Q1 and Q3 in our evaluation. Q1 has selections on the date type, and grouping and sorting on attributes with two characters. Q3 has two joins, selections on string and date types, and a grouping on three attributes.

Table IV. Queries of DBmbench

μSS	μNJ	μIDX
SELECT distinct (a3) FROM T1 WHERE Lo < a2 < Hi ORDER BY a3	SELECT avg (T1.a3) FROM T1, T2 WHERE T1.a1=T2.a1 AND Lo < T1.a2 < Hi	SELECT avg (a3) FROM T1 WHERE Lo < a2 < Hi

Table V. SQL Queries on Our Homegrown Data Sets

Name	Query
SEL	SELECT R.a1 FROM R WHERE Lo $\leq$ R.a1 $\leq$ Hi
AGG	SELECT max(R.a1) FROM R
ORD	SELECT R.a2 FROM R WHERE Lo $\leq$ R.a1 $\leq$ Hi ORDER BY R.a1
GRP	SELECT max(R.a1) FROM R GROUP BY R.a2
NEJ	SELECT R.a1 FROM R,S WHERE R.a1$\leq$S.a1 AND ... AND R.an$\leq$S.an AND Lo $\leq$ R.a2 $\leq$ Hi AND Lo$'$ $\leq$ S.a2 $\leq$ Hi$'$
EJ	SELECT R.a1 FROM R,S WHERE R.a1=S.a1

We used the standard data generator[6] from the TPC-H Web site to generate TPC-H data sets. We used the scale factors (SF) of 1 and 10 for the data sets, with total data sizes of about 1 and 10 GB, respectively. These two scales cover two cases. The data set with $SF = 1$ fits into the main memory, but exceeds the device memory, whereas the data set with $SF = 10$ is larger than both the main memory and the device memory.

We used our homegrown micro benchmarks as well as DBmbench [Shao et al. 2005] for detailed studies. DBmbench consists of three queries, including a scan-dominated query (μSS), a join-dominated query (μNJ) and an index-dominated query (μIDX). Table IV shows the queries in DBmbench. DBmbench consists of two relations, $T1$ and $T2$, each with three integer attributes: $a1$, $a2$ and $a3$; and the padding field. Attribute $a1$ of $T2$ has a reference key constraint with attribute $a1$ of $T1$. Following previous studies [Ailamaki et al. 1999; Shao et al. 2005], we set the length of "padding" to make a record 100 bytes long. The values of $a1$, $a2$, and $a3$ are uniformly distributed between 1 and 150,000, between 1 and 20,000, and between 1 and 50, respectively. The parameters in the predicate, Lo and Hi, are used to obtain different selectivities. In our study, we set the selectivity of μSS to be 10%, the join selectivity of μNJ to be 20%, and the selectivity of μIDX to be 0.1%. These settings for DBmbench represent the characteristics of TPC benchmarks' best [Shao et al. 2005].

Our homegrown workload contains various queries on relations R and S. Both tables contain n fields, $a_1, \ldots, a_n$, where a_i is a randomly generated 4-byte integer. The value range of a_2 is a parameter with a range $[0, A_2]$ for testing the group-by operator (the default value for A_2 is $|R|$), and the other fields with range of $[0, 2^{31} - 1]$. Additionally, we varied n to increase or decrease the tuple size.

Table V shows the SQL queries on our homegrown data sets. Our workloads use parameters in the predicate to control the selectivity. The queries include simple ones such as selection and aggregations, and complex ones such as joins.

[6]http://tpc.org/tpch/.

The selection queries in our own workloads have two forms. One is the range selection (Lo<Hi) and the other the point selection (Lo = Hi). We use table scans to evaluate SQL queries unless specified. Since we focused on the selection performance, we set (Hi-Lo) to a small constant, such as ten, in the non-equality predicate.

The join queries in our own workloads have two forms. One of the two join queries is an equi-join and the other a non-equijoin. The equi-join takes $R.a_1 = S.a_1$ as the predicate and the non-equijoin $R.a_1 < S.a_1 \ and ... and \ R.a_n < S.a_n$. All fields of each table are involved in the non-equijoin predicate so that an entire tuple is brought into the cache for the evaluation of the predicate. We used the non-indexed NLJ to evaluate the non-equijoin and used the indexed NLJ, the sort-merge join, or the hash join to evaluate the equi-join.

We run each experiment five times, and report the average value. Among these five runs, we observed stable performance results (with the coefficient of variance smaller than 5%).

—*Implementation details on the cost model*. On the CPU, we have adopted the CPU-based cost model in the previous study [Manegold et al. 2002]. On the GPU, we implemented micro benchmarks to calibrate the parameters in the cost model. The calibration is specific to the architectural features of the GPU. We measure the initialization overhead for invoking a memory transfer, T_0, to be the time for transferring one byte of data from the main memory to the device memory. The memory bandwidth $Band$ between the GPU device memory and the main memory is calibrated by transferring a large data array—one half of the device memory size. The coalesced and non-coalesced device memory bandwidths, B_h and B_l, are measured from the map primitive with and without the coalesced access optimizations, respectively. Note, the map primitive is with thread parallelism optimization. We run the calibration ten times and choose the average calibration value as the value used in the cost model. Through calibration, we obtain the following parameter values: $T_0 = 0.015$ ms, $Band = 3.1$ GB/sec, $B_h = 64$ GB/sec, $B_l = 32$ GB/sec, and the memory block size (b) is 256 bytes.

—*Implementation details on CPU*. For comparison, we have implemented highly optimized CPU-based primitives and query processing algorithms. These CPU-based parallel query processing algorithms are adopted from the OpenMP[7] implementation in the previous study [He and Luo 2008] and are rewritten using our optimized primitives. We use cache optimization techniques [Shatdal et al. 1994], to fine tune the performance of the parallel implementation. With these optimized primitives, we implement query operators, including four join algorithms—the blocked NINLJ [Shatdal et al. 1994], the INLJ with the CSS-tree index [Rao and Ross 1999], the SMJ with the optimized quick sort [LaMarca and Ladner 1997] and the radix HJ [Boncz et al. 1999]. Note that, due to the dynamic memory allocation capability of the CPU, the CPU-based algorithms do not use the three-step output scheme

[7]http://www.openmp.org/.

of the GPU-based algorithms. Each worker thread outputs its results in its local memory buffer, and the results in these memory buffers are merged into the final result. We compiled our implementation using MSVC 8.0 with full optimizations. In general, the parallel CPU-based primitives and query processing algorithms are 2–4x faster than the sequential ones on the quad-core CPU.

—*Implementation details on GPU*. We implemented our primitives and join algorithms using CUDA. CUDA is a GPGPU programming framework for recent NVIDIA GPUs. In the CUDA programming API, the developer can program the GPUs without any knowledge of graphics-rendering APIs. Similar abstractions are also available on AMD GPUs using their compute abstraction layer (CAL) API. Both of these APIs expose a general purpose, massively multi-threaded parallel computing architecture and provide a programming environment similar to multi-threaded C/C++. We use the same tuning on the parameters such as the number of threads for each thread group and the number of thread groups as the previous study [He et al. 2007, 2008].

—*Implementation details on coprocessing*. The coprocessing scheme uses the implementation from the CPU and the GPU ends. To hide the complexity of different processors, we implemented the CPU and the GPU ends as two DLLs (Dynamic Linked Libraries). We declared and implemented similar interfaces for the primitives and the relational operators in both DLLs. Our coprocessing scheme calls the interfaces provided in the DLLs without the knowledge of the processor. We studied the overhead of initializing the DLLs by comparing the performance of different implementations with and without wrapping the interfaces into DLLs, and found that the initialization overhead was negligible. Since the current GPU does not support multi-tasking, we add a lock on the GPU to ensure the synchronized accesses to the GPU. That is, before entering the code region for processing on the GPU, we acquire the lock, and release the lock once leaving the code region. In our experiments, we found that this locking overhead was negligible due to the large task granularity for the GPU.

When the input data is stored in the external storage, we perform the I/O operation with the assistance of the CPU, since current GPUs cannot directly access the data in the hard disk. We perform the file I/O on the CPU and load the file data chunk by chunk into a buffer in the main memory. The chunk size is bounded by the main memory.

6.2 TPC-H Performance Evaluation

Figure 7 shows the query plans of TPC-H queries Q1 and Q3 in GDB when $SF = 1$ and $SF = 10$. Each operator in the query plan is represented in one of the four execution modes as the combinations with or without partitioning, and CPU or GPU processing. Among these four execution modes, the CPU processing with partitioning is for data whose size is larger than the main memory, and the other three are GPU coprocessing modes of GDB.

Fig. 7. Query plans of evaluating Q1 and Q3 in TPC-H benchmark.

Table VI. Performance (in Seconds) of TPC-H Queries

$SF = 1$			$SF = 10$		
	Q1	Q3		Q1	Q3
DBMS X	14.0	3.8	DBMS X	859	1880
CPU	1.01	0.79	CPU	244.0	250.9
GDB	0.89	0.66	GDB	241.4	250.2

Table VI shows the performance of running the three queries. For comparison, we also show the performance of our CPU-based engine as well as as well as a commercial DBMS (denoted as DBMS X). The buffer pool size of DBMS X is set to be 1 GB. All the results are obtained in a warmed buffer, where we ran the query multiple times until the execution time of a query become stable. Both the CPU and the GDB engines outperform DBMS X by over 13.8 times and 3.5 times when $SF = 1$ and $SF = 10$, respectively, which demonstrates the efficiency of our implementation. The overall performance of GDB is only slightly faster than the CPU-based engine. Detailed studies on the time breakdown of both engines show that disk I/O time contributes 98% to the total execution time when $SF = 10$. In such I/O-dominant workloads, the GPU

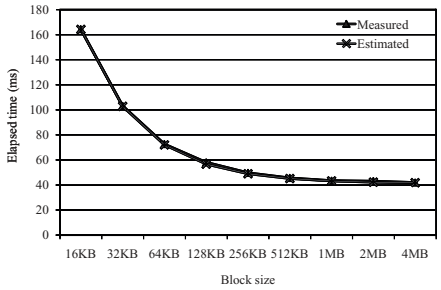


Fig. 8. Data transfer time from the main memory to the device memory ($\|R\| = 128$MB). When the block size is larger than 4MB, the peak bandwidth is 3.1 GB/sec.

acceleration has an insignificant impact on the overall performance. We further examine the computation time that is estimated by subtracting the I/O time from the total elapsed time in the TPC-H queries. When $SF = 10$, GDB takes 23% and 13% less computation time than the CPU-based engine on Q1 and Q3, respectively.

6.3 Micro-Level Performance Evaluation

We have two sets of experiments for micro-level evaluation of GDB. The first set of experiments is to evaluate our cost model, including data transfer between device memory and main memory and execution time on the GPU. The second set of experiments focuses on the performance evaluation using the micro benchmarks.

We varied the characteristics of the data sets in our homegrown micro benchmarks. By default, the measured results for non-indexed NLJs were obtained when $|R| = |S| = 1M$ and $r = s = 128$ bytes, and the measured results for other operations and queries were obtained when $|R| = |S| = 16M$ and $r = s = 8$ bytes.

6.3.1 *Cost Models. Data transfer between device memory and main memory*. Figure 8 shows the estimated and measured memory copy times from the main memory to the device memory. Similar results are obtained for data transfer from the device memory to the main memory. Given a certain block size, we transfer the data block by block. Due to the overhead associated with each transfer, both the measured and the estimated copy times increase as the block size decreases. When the block size is larger than 4 MB, the copy time remains almost constant. That means when the relation size is larger than 4 MB, the bandwidth is fully utilized. Our cost model on the memory transfer accurately predicts the measured performance.

Execution time on the GPU. Figure 9 shows the measured and estimated execution times of the four join algorithms on the GPU. The estimation methods include our model and the traditional model [Manegold et al. 2002], denoted as "Estimation" and "Estimation(old)," respectively. We fixed the tuple size to be 8 bytes and varied the number of tuples in both relations.

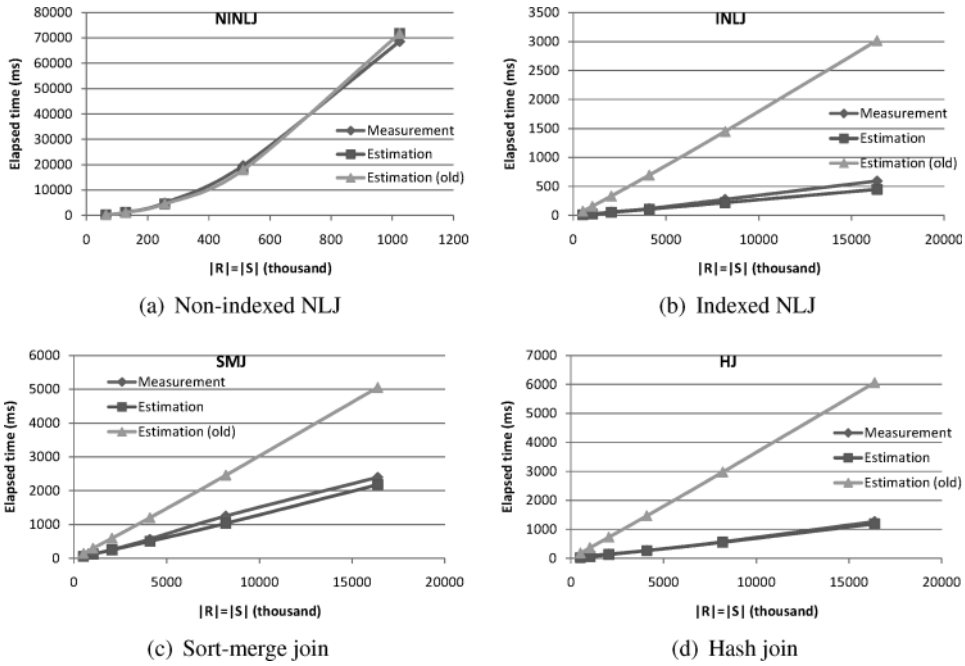


Fig. 9. Model validation on the execution time of the GPU.

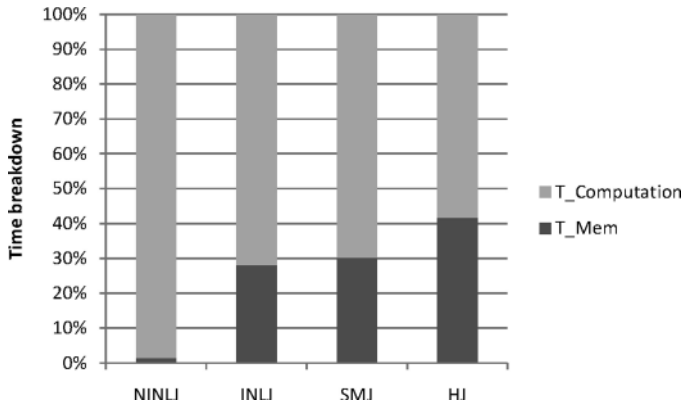


Fig. 10. The estimated time breakdown of the four joins for the GPU.

For NINLJ, the traditional approach and our model have the same estimations. For the other three join algorithms, including INLJ, SMJ, and HJ, the cost estimations with the traditional approach overestimates the elapsed time. In contrast, our model is more accurate than the model with the traditional approach on these three joins. This difference between the estimation accuracy for NINLJ and the other three joins can be explained from the estimated time breakdown of the four joins, as shown in Figure 10. In NINLJ, memory stalls are almost completely hidden, whereas in the other three joins memory

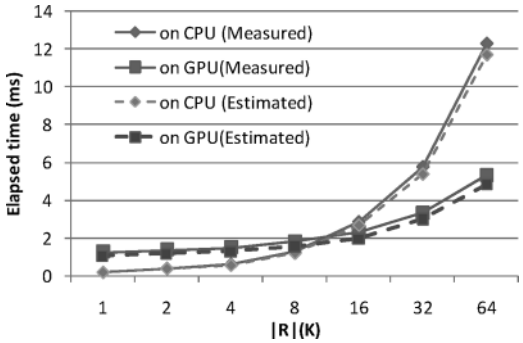


Fig. 11. The break-even point for GPU-based over CPU-based sort.

stalls constitute 30–40% of the elapsed time. Both models estimate computation highly accurately, but our estimation of memory stalls is relatively less accurate due to the runtime dynamics of memory accesses in a massively parallel execution environment. This difference between the estimation accuracy for computation versus memory stalls results in the difference between NINLJ versus the other three joins. It is also the reason for the traditional model to fail on estimating the cost for the three joins other than NINLJ.

Overall, our estimation approximates the measurement well for the four join algorithms. It correctly predicts the order of relative performance of the four join algorithms. For example, our model predicts that INLJ is the most efficient and NINLJ is the least efficient among the four join algorithms.

Break-even points for GPU processing. Our cost model is used to determine the break-even point between the GPU- and the CPU-based evaluations. Figure 11 shows the measured and estimated elapsed times for the GPU-based and the CPU-based quick sorts. When R contains 1000 tuples of a total size of 8 KB, the transfer time is over 40% of the total elapsed time of the GPU-based sort. This ratio decreases to 10% when $|R|$ is 256K. The break-even point for the quick sort is $|R| = 16K$. The calculation based on the cost model correctly predicts this break-even point.

6.3.2 *Processing Algorithms.* We perform detailed studies on processing algorithms. First, we evaluate individual processing algorithms in GDB, including primitives, access methods, and coprocessing schemes. In these studies, we focus on the elapsed time in the GPU-based operations and exclude the time of data transfer between the GPU and the CPU. These measurements are meaningful since GPU-based primitives and operators are executed in a query after their input data are already in the device memory and their output data are stored in the device memory as input to subsequent primitives and operators.

Second, we measure the end-to-end performance of the queries in the micro benchmarks. For both CPU- and GPU-based algorithms, we measure the elapsed time from the moment of issuing the query to the moment of obtaining the query result.

Primitives. We studied the three optimization techniques on the GPU: the thread parallelism, the coalesced access, and local memory optimizations. We

Table VII. Elapsed Time for Primitives and Joins (ms)

Primitive	CPU	GPU	Speedup
Map	109	4	27.3
Scatter	1312	104	12.6
Gather	1000	103	9.7
Prefix scan	141	14	10.1
Reduce	31	11	2.8
Filter	62	37	1.7
Split	813	125	6.5
Sort(qsort)	2313	945	2.4

Operators	CPU	GPU	Speedup
Selection	63	36.08	1.7
Projection	20	0.86	23.3
OrderBy	2500	1000	2.5
GroupBy	2323	945	2.5
Aggr.	32	11.62	2.8
NINLJ	528.0×10^3	75.0×10^3	7.0
INLJ	4235	649	7.0
SMJ	5030	1946	2.6
HJ	2550	1327	1.9

obtained similar results on the optimization techniques as previous studies [He et al. 2008].

Table VII shows the performance comparison between the CPU- and GPU-based optimized primitives and joins. We define the *speedup* to be the ratio of the execution time on the CPU to that on the GPU. Overall, the GPU-based algorithms achieve a performance speedup of 2–27x over the CPU-based algorithms. We obtained similar performance speedup with different data sizes.

We have the following observations. First, the average bandwidth of the optimized map primitive is 2.4 GB/sec and 64 GB/sec on the CPU and the GPU, respectively. The speedup of the optimized GPU map is 27x over the CPU-based map. Additionally, it has a high bus utilization of 75%, given the theoretical bandwidth of 86 GB/sec. Second, the scatter and the gather have much lower bandwidths than the map due to their random access nature. Third, in the split on both the GPU and the CPU, the scatter takes over 70% of the total execution time. Fourth, the speedup of the GPU-based quick sort algorithm is 2x over the optimized quick-sort on the quad-core CPU. Comparing the two GPU-based sorting algorithms, we find that the quick sort is around 30% faster than the bitonic sort (the result on bitonic sort is not shown in the table). This result is consistent with the fact that the quick sort has a lower complexity than the bitonic sort. We used the quick sort as our sorting primitive in the CUDA implementation. Fifth, the GPU-based operators achieve a performance speedup of 1.7–23.3x over their CPU-based counterparts due to the highly efficient primitives. In particular, the GPU-based NINLJ and INLJ are over six times faster than their CPU-based counterparts, because their computation maps well to the GPU architecture.

Handling various data types. To demonstrate the feasibility of handling various data types in GDB, we studied the performance of the GPU- and the CPU-based filters on four different data types, namely *integer*, *float*, *date*, and *string*. The first three types are of a fixed length, and strings have variable sizes (with an average length of eight). Figure 12 shows performance comparisons when the number of tuples is 16 million, and the filter selectivity is around 1%. On these four data types, the GPU-based filter is around 1.6–5.5x faster than its CPU-based counterpart. We also varied the average length of the string, and got similar performance speedups.

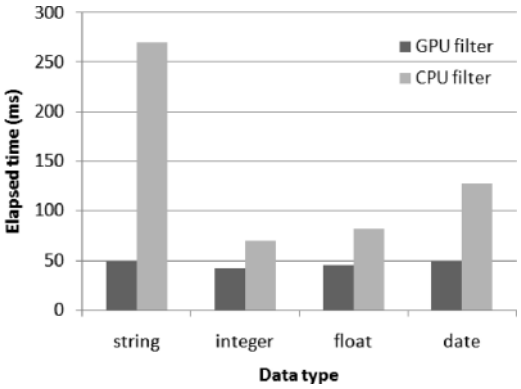
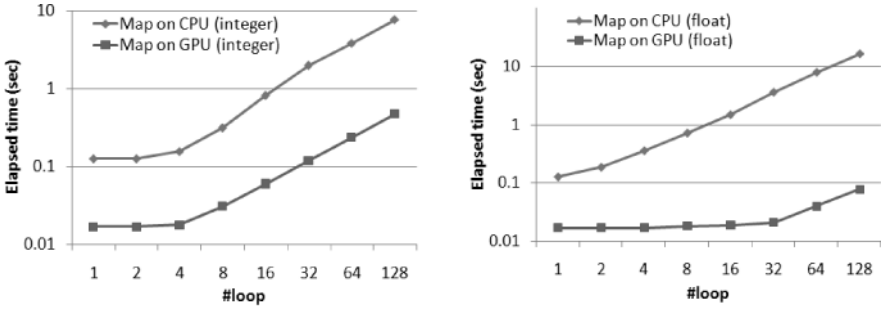


Fig. 12. The performance of filters with different data types.



(a) Map with integers (b) Map with floats

Fig. 13. The performance of map with the number of arithmetic operations varied.

Comparing the performance of numeric types on the GPU and the CPU, we find that GDB exploits the high floating point computation capability of the GPU. We evaluated a map primitive with a map function consisting of multiple arithmetic operations including multiplications and divisions. We varied the amount of computation by the number of arithmetic operations in a map. Figure 13(a,b) shows the performance of the GPU- and the CPU-based maps with integers and floats, respectively. The GPU-based maps outperform their CPU-based counterparts by over an order of magnitude. As the number of arithmetic operations increases, the performance speedup is stable for integers, and increases for floats. Also, we observe that the GPU-based map with floats is faster than that with integers. This indicates that floating point computation fits better than integers on the GPU.

Access methods. We construct the tree index and the hash index on the field a_1 of R. We evaluated our access methods including the B+-tree and the hash index using a number of selection queries. Figures 14(a) and (b) show the performance comparison of evaluating SEL the tree search and the hash search on the CPU and the GPU. Each SEL instance is with Lo and Hi equal to a random generated integer. For each access method, we fixed the number of searches to be one million and varied the number of records in the index. The GPU-based tree

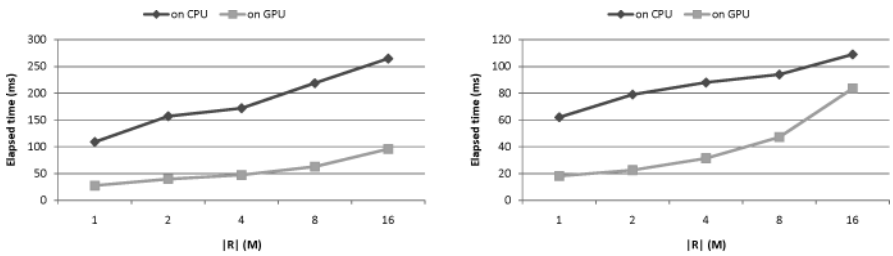


Fig. 14. Performance comparison of the CPU- and the GPU-based access methods.

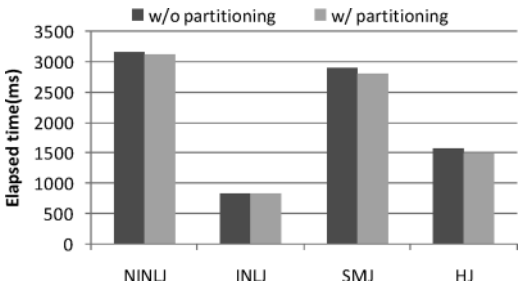


Fig. 15. Performance of evaluating an Operator with Partitioning on both CPU and GPU.

searches and hash searches are 3.5 and 2.6 times faster than their CPU-based counterparts.

Coprocessing with data partitioning between CPU and GPU. We studied the performance impact of evaluating an operator with partitioning on both processors. Figure 15 shows the elapsed time of the four joins in our homegrown queries with and without partitioning. The evaluation without partitioning is performed on the GPU only. The performance impact of partitioning is insignificant on the four joins. The performance improvement ratio is smaller than the ratio of the overall processing power (CPU+GPU) divided by the GPU or the CPU.

We analyzed the performance of the four joins in two cases. First, in NINLJ and INLJ, the GPU-based algorithms are much faster than their CPU-based counterparts, and the partitioning overheads are relatively small. As such, involving the CPU helps little. Second, in SMJ and HJ, the CPU gets 25% and 30% of data for processing respectively, but the performance gain is still insignificant. Our detailed studies show that the runtime overhead of partitioning contributes to over 20% of the total elapsed time for these two joins. The performance improvement of partitioning is offset by the run-time overhead of the partitioning. We have conducted experiments on other operators, and also observed insignificant performance improvement by partitioning.

Queries in micro benchmarks. Table VIII shows the performance of evaluating the queries in our homegrown workload and DBmbench, respectively. We use INLJ, SMJ and HJ to evaluate the EJ query, denoted as EJ(INLJ), EJ(SMJ), and EJ(HJ), respectively. For comparison, we show the performance

Table VIII. Elapsed Time of Evaluating the Queries on Micro Benchmarks (ms)

	CPU_ONLY	GPU_ONLY	Coprocessing	$Speedup_L$	$Speedup_H$
SEL(point)	47	375	48	0.98	7.81
SEL(range)	172	407	175	0.98	2.33
AGG	16	266	16	1.00	16.63
ORD	3609	1735	1703	1.02	2.12
GRP	3984	1797	1594	1.13	2.50
NEJ	45578	3906	3828	1.02	11.91
EJ(INLJ)	4000	891	880	1.01	4.54
EJ(SMJ)	5562	2809	2709	1.04	2.05
EJ(HJ)	3640	1687	1642	1.03	2.21
uSS	62	219	63	0.98	3.48
uNJ	1828	500	455	1.10	4.02
uIDX	79	125	78	1.01	1.60

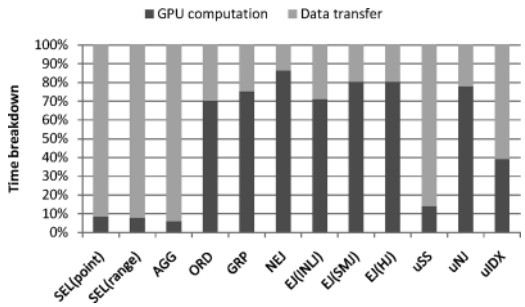


Fig. 16. Time breakdown for micro benchmarks with GPU_ONLY.

of evaluating the queries with the CPU only ("CPU_ONLY"), the GPU only ("GPU_ONLY"), and coprocessing ("Coprocessing"). For each query, let the total elapsed time, including data and result transfer time, of CPU_ONLY, GPU_ONLY, and Coprocessing to be t_1, t_2, and t_c, respectively. We calculate $Speedup_L$ and $Speedup_H$ for Coprocessing to be $\frac{min(t_1,t_2)}{t_c}$ and $\frac{max(t_1,t_2)}{t_c}$, respectively. $Speedup_H$ is the performance gain of our cost-based coprocessing over that when the wrong processor is chosen, and $Speedup_L$ for the performance gain over the better scheme between CPU_ONLY and GPU_ONLY.

Comparing the performance of CPU_ONLY and GPU_ONLY, we find that, for simple queries, the GPU-based algorithms are much slower than their CPU-based counterparts. This is mainly because data transfer time between the main memory and the device memory is the major bottleneck for the GPU-based algorithms evaluating these simple queries. Figure 16 shows the time breakdown of processing the queries with GPU_ONLY, dividing the total execution time of a GPU-based algorithm into two components including the total time for copying input data into the device memory and result output to the main memory, and the GPU computation time. For complex queries, the GPU-based algorithms are faster than their CPU-based counterparts. The data transfer time between the main memory and the device memory is insignificant for evaluating these queries.

For all queries, $Speedup_L$ is between 0.98 and 1.13 (most of them are larger than one), and $Speedup_H$ is between 1.60 and 16.63. $Speedup_L$ is slightly smaller than one on simple queries such as SEL, due to the overhead of cost estimation. Mostly, Coprocessing achieves a performance similar to or better than than the best one of processing on the GPU only and the CPU only. Based on the cost models, Coprocessing chooses suitable processors to execute an operator. For instance, the NEJ query with Coprocessing is slightly faster than both the CPU_ONLY and the GPU_ONLY algorithms, through choosing the CPU to evaluate the selections and coprocessing with partitioning to evaluate the NINLJ on the selection results. Compared with the CPU_ONLY algorithm, Coprocessing offloads the NINLJ to the GPU. Compared with the GPU_ONLY algorithm, Coprocessing performs the selection on the CPU and reduces the memory transfer between the device memory and the main memory.

6.4 Summary

In summary, with GPU acceleration, GDB achieves a significant performance improvement on memory-resident data and has a comparable performance to our optimized CPU-based engine on disk-based data. Specifically, on TPC-H benchmark queries with data sets larger than the memory, GPU coprocessing reduces the computation time up to 23%, even though the overall performance improvement is insignificant due to the disk I/O bottleneck. In contrast, on in-memory data our GPU-based primitives and query processing algorithms achieve a speedup of 2–27x over their optimized CPU-based counterparts. With data transfer time between the CPU and the GPU included, our GPU-based algorithms achieve a 2–7x performance speedup over their CPU-based counterparts for complex queries such as joins, and are 2–4x slower than their CPU-based counterparts for simple queries such as selections. With the cost-based GPU coprocessing scheme, the overall performance is similar to or better than using the CPU or using the GPU only.

7. DISCUSSION

The main motivation for studying GPU coprocessing on relational databases is that GPUs have a highly parallel hardware architecture that fits extremely well with data-parallel query processing. Devoting most die area to computation units, the GPU has a high computation capability, and is suitable for computation-intensive applications. Additionally, the device memory has a high bandwidth, and the massive thread parallelism and the fast on-chip local memory on the GPU hides the memory latency well. These architectural features can produce significant performance gains when using the GPU as a coprocessor.

While our study focuses on utilizing the GPU processor and device memory for query coprocessing on memory-resident data, we have made initial efforts on GPU coprocessing of disk-based data. As expected, the GPU is under-utilized on disk-based data. Nevertheless, increases in the main memory as well as device memory will help further utilize the computation power of the GPU. One trend is that memory-resident databases are getting even more significant: as memory capacities increase, most frequently accessed data items in relational

databases are likely to fit into the main memory. Another trend in GPU hardware development is that the device memory size is increasing as well, for example, the NVIDIA Tesla S1070 GPU has a 16 GB device memory.

Although there have been quite a few studies on accelerating individual operators using the GPU [Lieberman et al. 2008; He et al. 2008], we find implementing a full-fledged query on the use of a coprocessor on the GPU a challenging task. In particular, we have identified the following challenging issues for implementing query coprocessing on the current GPU. While these issues are challenging at the current status, they may be addressed with the combined efforts from hardware vendors and developers. For example, we previously considered that supporting high precision numbers on the GPU was challenging [He et al. 2008]. Current GPUs have already supported double-precision floating point numbers, which addresses that challenge.

First, with the exposure of the massively multi-threaded hardware architecture on the GPU, the complexity of GPU programming becomes a major hurdle for developing correct and efficient GPU programs. While current debugging facilities on the GPU can reduce the debugging effort, it is still tedious to effectively find bugs such as concurrency bugs in the massively multi-threaded implementation.

Second, while pipelined execution parallelism is attractive on the CPU for its small memory footprint, finer-grained pipelined execution, such as on a per-tuple basis, on the GPU would be inefficient. The reason is that current GPUs do not support dynamic memory allocation within kernel execution, and therefore we need to precompute the result size for each chunk of processing. In other words, the pipelined execution needs to be synchronized for each data chunk, which hurts the original pipelined parallelism. Note that our GPU query coprocessing algorithm can be considered as a special kind of pipelined processing with a large data chunk granularity that is bounded by the GPU memory capacity.

Third, the data transfer between the GPU device memory and the main memory can be a performance bottleneck for GPGPU applications. In our experiments, processing simple queries on the GPU can be much slower than the CPU-based algorithm due to the cost of this data transfer. Currently, we use a cost model to determine whether the cost of the data transfer between the GPU device memory and the main memory can be compensated by the performance gain of GPU processing over CPU processing. As future work, we plan to investigate compression techniques [Zukowski et al. 2006; Abadi et al. 2006] in GDB to reduce the overhead of data transfer between the main memory and the GPU memory. Additionally, since recent GPUs support asynchronous data transfer between the GPU memory and the main memory, we can utilize this support to hide the stall with useful computation on the GPU or on the CPU. Note, our cost model needs to adapt to the overlaps between the computation and the data transfer.

Finally, current GPUs do not support multitasking, and therefore execute one query at a time. This execution model together with the SIMD processing style makes the GPU unable to efficiently execute concurrent queries to multiple users. One possible alternative to having GPU coprocessing of multiple

concurrent queries is to let the CPU batch up queries into a single GPU kernel for execution. We hope that future GPUs will add support for multitasking, which will subsequently allow a more thorough study of GPU-based concurrent execution of multiple queries.

Addressing the challenges, we find the following two methodologies most effective for developing GDB. These two methodologies are essential for parallel programming, and should be taken into account for other general parallel computing platforms such as OpenCL.[8]

First, the primitive-based methodology has a high flexibility for GPU programming. With the divide-and-conquer nature, the primitive-based methodology breaks the query processing algorithms into a set of simple primitives, which allows us to tune and optimize them independently, thus reducing the programming complexity. Moreover, these primitives can be used to develop higher-level primitives and other applications. Additionally, the primitive-based approach allows us to easily replace the existing implementation of a certain primitive with a more efficient one whenever applicable.

Second, the cost-based approach is essential for the effectiveness of GPU coprocessing. We develop calibration-based and analytical cost models to decide whether to offload the task to the GPU. Unlike previous work focusing on the cost of individual operations such as scatter and gather [He et al. 2007], our cost models are sufficiently general in estimating the cost in three parts: the data transfer between the main memory and the device memory, the accesses to the GPU device memory, and the GPU computation. We believe our models can be adopted to other GPU coprocessing tasks, and other data types such as double-precision floating numbers on more recent GPUs.

8. CONCLUSION

Graphics processors have become an attractive alternative for general purpose high performance computing on commodity hardware. The continuing advances in hardware and the recent improvements on programmability make GPUs a promising platform for database query processing. In this study, we present our design, implementation, and evaluation of our GPU-based relational query processor, GDB. To the best of our knowledge, GDB is the first GPU-based relational query processor for main memory databases. Furthermore, we propose coprocessing techniques that take into account both the computation resources and the GPU-CPU data transfer cost so that each operator in a query can utilize a suitable processor, either the CPU or the GPU, for optimized overall performance. We have performed experiments on both memory-resident micro benchmarks and disk-resident TPC-H workloads. Our results show that our GPU-based algorithms are 2–27x faster than their optimized CPU-based counterparts on in-memory data. Moreover, the performance of our coprocessing scheme is similar to or better than both the GPU-only and the CPU-only schemes.

The GDB packages are available on our project site.[9]

[8]http://www.khronos.org/opencl/.

[9]http://www.cse.ust.hk/gpuqp.

REFERENCES

ABADI, D., MADDEN, S., AND FERREIRA, M. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, NY, 671–682.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 266–277.

AILAMAKI, A., GOVINDARAJU, N., HARIZOPOULOS, S., AND MANOCHA, D. 2006. Query co-processing on commodity processors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

BANDI, N., SUN, C., AGRAWAL, D., AND ABBADI, A. E. 2004. Hardware acceleration in commercial databases: a case study of spatial operations. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 1021–1032.

BLELLOCH, G. E. 1990. Prefix sums and their applications. Tech. rep. CMU-CS-90-190, Carnegie Mellan University.

BLYTHE, D. 2006. The direct3d 10 system. SIGGRAPH, ACM Press, NY.

BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 54–65.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for gpus: Stream computing on graphics hardware. SIGGRAPH, ACM Press, NY.

CIESLEWICZ, J., BERRY, J., HENDRICKSON, B., AND ROSS, K. A. 2006. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware (DaMoN)*. ACM, New York.

CIESLEWICZ, J. AND ROSS, K. A. 2007. Adaptive aggregation on chip multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 339–350.

DEWITT, D. AND GRAY, J. 1992. Parallel database systems: the future of high performance database systems. *Comm. ACM 35*, 6, 85–98.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, Washington, DC, 285.

GEDIK, B., BORDAWEKAR, R. R., AND YU, P. S. 2007. Cellsort: High performance sorting on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 1286–1297.

GEDIK, B., YU, P. S., AND BORDAWEKAR, R. R. 2007. Executing stream joins on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 363–374.

GOLD, B., AILAMAKI, A., HUSTON, L., AND FALSAFI, B. 2005. Accelerating database operators using a network processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN)*. ACM, New York, NY, 1.

GOVINDARAJU, N., GRAY, J., KUMAR, R., AND MANOCHA, D. 2006. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 325–336.

GOVINDARAJU, N., LLOYD, B., WANG, W., LIN, M., AND MANOCHA, D. 2004. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 215–226.

GOVINDARAJU, N. K., RAGHUVANSHI, N., AND MANOCHA, D. 2005. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, 611–622.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2, 73–169.

HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. 2008. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 981–992.

HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A. 2007. CUDPP: CUDA data parallel primitives library.

HE, B., GOVINDARAJU, N. K., LUO, Q., AND SMITH, B. 2007. Efficient gather and scatter operations on graphics processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, 1–12.

HE, B. AND LUO, Q. 2008. Cache-oblivious databases: Limitations and opportunities. *ACM Trans. Datab. Syst. 33*, 2, 1–42.

HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. 2008. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 511–524.

HÉMAN, S., NES, N., ZUKOWSKI, M., AND BONCZ, P. 2007. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware (DaMoN)*. ACM, New York, NY, 1–6.

HONG, W. AND STONEBRAKER, M. 1991. Optimization of parallel query execution plans in xprs. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*. IEEE Computer Society Press, Los Alamitos, CA, 218–225.

JOHNSON, R., HARIZOPOULOS, S., HARDAVELLAS, N., SABIRLI, K., PANDIS, I., AILAMAKI, A., MANCHERIL, N. G., AND FALSAFI, B. 2007. To share or not to share? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 351–362.

LAMARCA, A. AND LADNER, R. E. 1997. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 370–379.

LIEBERMAN, M. D., SANKARANARAYANAN, J., AND SAMET, H. 2008. A fast similarity join algorithm using graphics processing units. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

LIU, B. AND RUNDENSTEINER, E. A. 2005. Revisiting pipelined parallelism in multi-join query processing. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 829–840.

LU, H., TAN, K.-L., AND SAHN, M.-C. 1990. Hash-based join algorithms for multiprocessor computers with shared memory. In *Proceedings of the 16th International Conference on Very Large Databases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 198–209.

MANEGOLD, S., BONCZ, P., AND KERSTEN, M. 2002. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*.

NGUYEN, H. 2008. *GPU gems 3*. Addison-Wesley.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum 26*.

RAO, J. AND ROSS, K. A. 1999. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 78–89.

SCHNEIDER, D. A. AND DEWITT, D. J. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD Rec. 18*, 2, 110–121.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM Press, New York, NY, 23–34.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*.

SHAO, M., AILAMAKI, A., AND FALSAFI, B. 2005. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, 254–267.

SHATDAL, A., KANT, C., AND NAUGHTON, J. F. 1994. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 510–521.

STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. 2005. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 553–564.

SUN, C., AGRAWAL, D., AND ABBADI, A. E. 2003. Hardware acceleration for spatial selections and joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 455–466.

TARDITI, D., PURI, S., AND OGLESBY, J. 2006. Accelerator: using data parallelism to program GPUS for general-purpose uses. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

ZHOU, J., CIESLEWICZ, J., ROSS, K. A., AND SHAH, M. 2005. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 49–60.

ZUKOWSKI, M., HEMAN, S., NES, N., AND BONCZ, P. 2006. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Washington, DC, 59.

Received August 2008; revised March 2009, July 2009; accepted August 2009