

# Relational Joins on Graphics Processors

Bingsheng He, Ke Yang<sup>#</sup>, Rui Fang<sup>§</sup>, Mian Lu,  
Naga K. Govindaraju<sup>\*</sup>, Qiong Luo, and Pedro V. Sander

Hong Kong University of Science and Technology, China  
{saven, lumian, lu, psander}@cse.ust.hk

<sup>§</sup>Highbridge Capital Management LLC, USA  
rui.fang@highbridge.com

<sup>#</sup>Zhejiang University, China  
yk.cadcg@gmail.com

<sup>\*</sup>Microsoft Corporation, USA  
nagag@microsoft.com

## ABSTRACT

We present a novel design and implementation of relational join algorithms for new-generation graphics processing units (GPUs). The most recent GPU features include support for writing to random memory locations, efficient inter-processor communication, and a programming model for general-purpose computing. Taking advantage of these new features, we design a set of data-parallel primitives such as split and sort, and use these primitives to implement indexed or non-indexed nested-loop, sort-merge and hash joins. Our algorithms 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. We have implemented our algorithms on a PC with an NVIDIA G80 GPU and an Intel quad-core CPU. Our GPU-based join algorithms are able to achieve a performance improvement of 2-7X over their optimized CPU-based counterparts.

**Categories and Subject Descriptors:** H.2.4 Systems, Query processing; Relational databases

**General Terms:** Algorithms, Measurement, Performance.

**Keywords:** relational database, join, sort, primitive, parallel processing, graphics processors

## 1. INTRODUCTION

Graphics processing units (GPUs) are specialized architectures traditionally designed for gaming applications. Recent research has shown that they can significantly speed up database query processing [5][14][15][16][36]. Moreover, new generation GPUs, such as AMD R600 and NVIDIA G80, have transformed into powerful co-processors for general-purpose computing (GPGPU). In particular, they provide general parallel processing capabilities, including support for scatter operations and inter-processor communication, as well as general-purpose

programming languages such as NVIDIA CUDA [27]. In this paper, we investigate the design and implementation of common relational join algorithms on such GPUs.

Joins are the cornerstone operator in relational database systems and CPU-based join algorithms have been studied extensively in the literature. Basic join algorithms include non-indexed and indexed nested-loop joins (NINLJ and INLJ respectively), the sort-merge join (SMJ) and the hash join (HJ). Many variants have been designed for in-memory databases [10][31][34] and for parallel databases [13][24][32]. These studies have shown that the implementation techniques, as well as the design, have a great impact on the join performance on CPU-based architectures. In general, memory stalls are a major performance factor for CPU-based relational joins [10][34].

Similar to CPUs, in particular multi-core CPUs, GPUs are commodity hardware consisting of multiple processors. However, these two types of processors differ significantly in their hardware architecture. Specifically, GPUs provide parallel lower-clocked execution capabilities on over a hundred SIMD (Single Instruction Multiple Data) processors whereas current multi-core CPUs typically offer out-of-order execution capabilities on a much smaller number of 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. These GPU hardware design choices provide higher computational capabilities, better latency tolerance and higher memory bandwidth.

We explore how relational joins can utilize hardware features of the GPU. In particular, the SIMD design and the massively multithreaded capability in GPUs require our algorithms to achieve good load balancing across processors to hide the latency effectively. Moreover, most GPUs lack hardware support for handling read/write conflicts among concurrent threads. On one hand, this design choice reduces the hardware complexity. On the other hand, high-level abstractions and carefully designed patterns in the software are necessary for correctness and efficiency.

Considering the characteristics of GPUs and individual join algorithms, we design a set of data-parallel primitives that are used as building blocks for our join algorithms. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SIGMOD'08*, June 9-12, 2008, Vancouver, BC, Canada.

Copyright 2008 ACM 978-1-60558-102-6/08/06...\$5.00.

local memory, and (3) the coalesced access. Specifically, our map primitive employs the coalesced accesses among GPU threads to fully utilize the video memory bandwidth; our split operation avoids the read/write conflicts by aligning histograms to the GPU threading architecture efficiently; our scatter and gather operations work in multiple passes for improved spatial locality in the memory access; and our sort algorithm uses the map primitive to implement a sorting network, or uses the split primitive to implement a quick sort.

Utilizing this small set of data-parallel primitives, we have designed and implemented GPU-based algorithms for NINLJ, INLJ, SMJ, and HJ. Specifically, our NINLJ is block-nested loops, with a data block mapped to a group of threads within a processor; our INLJ constructs a GPU-based variant of the CSS-Tree (Cache-Sensitive Search Trees) [31] and performs a massive number of concurrent index searches in the join; our SMJ utilizes quantiles for balanced range-partitioning and merges sorted partitions in parallel; and our HJ recursively splits the relation into multiple partitions and performs joins on the matching partitions in parallel. We have implemented all of our GPU-based primitives and join algorithms using CUDA [27], NVIDIA’s GPGPU language, and DirectX 10 [6], a graphics API for programmable GPUs. We evaluated our GPU-based algorithms in comparison with their optimized parallel counterparts on an Intel quad-core CPU. All join algorithms operate on memory-resident data organized in the column-based model [10][35].

In summary, this paper makes the following three contributions. First, we identify the technical challenges in performing parallel query processing on GPUs and provide general solutions to address these challenges. Our GPU-based data-parallel primitives are applicable to not only joins but also other query operators. Second, we design and implement several representative join algorithms on the new-generation GPUs and empirically evaluate these algorithms in comparison with the optimized CPU-based join algorithms. To the best of our knowledge, this is the first attempt to develop relational joins on graphics processors. Third, we discuss the lessons we have learned from experience and provide insights and suggestions on GPU programming for the GPGPU and database communities.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the GPU architecture and review GPU- and CPU-based query processing techniques and parallel join algorithms. In Section 3, we describe the technical challenges of performing parallel query processing on GPUs, and present our solutions. These solutions are then used as building blocks for our join algorithms, which are described in Section 4. We experimentally evaluate our algorithms in Section 5. We discuss the lessons learned from our experience in Section 6, and conclude in Section 7.

## 2. PRELIMINARY AND RELATED WORK

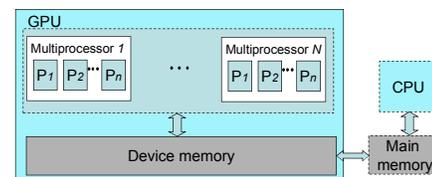
In this section, we introduce the GPU architecture and discuss related work.

### 2.1 Graphics Processors (GPUs)

GPUs are widely available as commodity components in modern machines. They are used as co-processors for the CPU [1]. GPU programming languages include graphics APIs such as OpenGL [28] and DirectX [6], and GPGPU languages such as NVIDIA CUDA [27], AMD CTM [2], Brook [8] and Accelerator [37].

With these APIs, programmers write two kinds of code, the kernel code and the host code. The host code runs on the CPU to control the data 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. A 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.

The GPU architecture model is illustrated in Figure 1. Such architecture is a common design for both AMD [2][7] and NVIDIA GPUs [27]. At a high level, the GPU consists of many SIMD multi-processors. At any given clock cycle, each processor of a multiprocessor executes the same instruction, but operates on different data. The GPU has a large amount of device memory, which has high bandwidth and high access latency. For example, the G80 GPU has an access latency of 200 cycles and the memory bandwidth of 86 GB/second. Additionally, each multiprocessor usually has a fast on-chip local memory, which is shared by all the processors in a multi-processor. The size of this local memory is small and the access latency is low.



**Figure 1. The GPU architecture model. The GPU is a co-processor to the CPU. It consists of multiple SIMD multiprocessors, and has a large amount of device memory. This model is applicable to both AMD’s CTM [2][7] and NVIDIA’s CUDA [27].**

GPU threads are different from CPU threads in that they have low context-switch and low creation time as compared to their CPU counterparts. On the GPU, threads on each multiprocessor are organized into *thread groups*. These thread groups are dynamically scheduled on the multiprocessors. Threads within a thread group share computation resources such as registers on a multiprocessor. Moreover, when multiple threads in a thread group access consecutive memory addresses, these memory accesses are grouped into one access. This hardware feature is called coalesced access.

### 2.2 Query Processing on GPUs

Recently, GPUs have been used to accelerate scientific, geometric, database and imaging applications. For an overview on the state-of-the-art GPGPU techniques, we refer the reader to the recent survey by Owens et al. [30]. We now briefly survey the techniques that use GPUs to improve the performance of database operations.

Sun et al. [36] used the rendering and search capabilities of GPUs for spatial selection and join operations. Bandi et al. [5] implemented GPU-based spatial operations as external procedures to a commercial DBMS. Govindaraju et al. presented novel GPU-based algorithms for relational operators including selections, aggregations [15] as well as sorting [14], and for data mining operations such as computing frequencies and quantiles for data

streams [16]. The existing work mainly develops OpenGL/DirectX programs to exploit the specialized hardware features of GPUs. In contrast, we focus on GPU-based algorithms for the join operation, which is a core operator in relational databases. Moreover, our algorithms are based on a many-core SIMD architecture model of the GPU, and thus can be applied to CPUs of a similar architecture. Based on a similar model, Sengupta et al. [33] implemented the segmented scan using the scatter. He et al. [19] proposed a multi-pass scheme to improve the scatter and the gather operations on the GPU. Our algorithms utilize these operations as primitives to compose join algorithms. Most recently, Lieberman et al. [23] implemented a similarity join using CUDA.

### 2.3 In-Memory Query Processing on CPUs

Memory stalls are an important factor for the overall performance of relational query processing [10][34]. Cache-conscious techniques have been the leading approach to improve the memory performance of the CPU joins.

Shatdal et al. [34] proposed the blocked NINLJ algorithm by applying cache blocking on the nested-loop join. In comparison, we determine the block size in NINLJ by the size of the local memory. Rao et al. [31] proposed a cache-optimized B+-tree index, namely the CSS-tree. A CSS-tree has a node size equal to the cache block size. Each node is fully packed with keys. Pointers are eliminated by laying out nodes contiguously, level by level. Index search is done through address arithmetic. We adopt this tree index to the GPU and optimize its performance by fitting the top levels of the tree index into the local memory. Lamarca et al. [22] studied the cache performance for the quick sort and showed that cache optimizations can significantly improve the overall performance. In comparison, we implement the quick sort on the GPU and use bitonic sort to sort partitions that fit into the local memory. Boncz et al. [10] proposed the radix hash join with a multi-pass partitioning method in order to optimize the cache performance. Our GPU-based hash join is a parallel version of the radix hash join with optimizations for the local memory.

With the same goal of reducing memory stalls, our local memory optimization aims at improving the spatial locality and temporal locality of the data accesses. In contrast with the hardware-managed cache on the CPU, our techniques are specifically designed for the local memory on GPUs, which is manipulated by the programmer and is shared by multiple threads.

### 2.4 Parallel Joins

Parallel algorithms greatly improve the performance of the relational join in shared-nothing systems [24][32] or shared-memory systems [11][25].

Liu et al. [24] investigated the pipelined parallelism for multi-join queries. In comparison, we focus on exploiting the parallelism within a single join operation. For a single join, Lu et al. [25] studied four hash-based join algorithms on a shared-memory multiprocessor system. Schneider et al. [32] evaluated one sort-merge and three hash-based join algorithms in a shared-nothing system. In the presence of data skews, techniques such as bucket tuning [32] and partition tuning [21] are used to balance loads among processor nodes. Azadegan et al. [3][4] used machine-specific communication primitives to develop parallel join algorithms on the SIMD Connection Machine (CM-2). Recently, Cieslewicz et al. [11] implemented a multi-threaded hash join

using the atomic operations supported in the Cray MTA-2 architecture.

In comparison with previous parallel join algorithms, our GPU-based parallel join algorithms take into account the GPU architectural characteristics and provide general, yet efficient solutions. Specifically, in contrast with using machine-specific primitives [3][4], we develop software primitives that are general and highly scalable for GPUs. Additionally, our thread parallelism does not require hardware-supported atomic operations.

## 3. PRIMITIVES

Based on the GPU architectural model, we have identified three technical challenges in join processing on GPUs:

- How to efficiently utilize both the computation resource and the memory bandwidth of the GPU, and to use parallel computation to hide memory latency. This challenge is critical in that joins are both computation and data intensive. Even though the GPU has massive thread parallelism and high memory bandwidth, its memory latency is also high. Therefore, we need to examine individual join algorithms and develop common building blocks that improve data parallelism.
- How to handle read/write conflicts efficiently. Since we do not have hardware-supported atomic operations for conflict handling, we need to develop an efficient conflict handling mechanism that is suitable for GPUs.
- How to handle data skews on GPUs. As on any parallel architecture, data skews must be handled effectively to balance the workload among processors so as to improve the overall performance.

We address these challenges in primitives, a small set of common operations that we design for join processing on the GPU. These primitives exploit the hardware features of the GPU and can be used for database query processing, including joins.

**Notation.** In this paper, we consider a join on two relations  $R$  and  $S$  with a single join attribute. We assume the join attribute to be an integer for simplicity.  $R[i]$  represents the  $i$ th tuple of  $R$ . The notations used throughout this paper are summarized in Table 1.

**Table 1. Notations used in this paper**

Parameter	Description
$Bp$	Total number of thread groups on the GPU
$T$	Number of threads per thread group
$M$	The size of local memory per thread group
$R, S$	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\ $	Sizes of $R$ and $S$ (bytes)

### 3.1 Baseline Design

We aim at designing and implementing a complete set of parallel primitives for relational query processing. In this section, we describe our primitives, namely map, scatter, gather, prefix scan, split and sort. These primitives are used as constructs for our join algorithms and have the following features:

- 1) They have low synchronization overhead, thus achieving close to peak performances on GPUs.
- 2) They are scalable to hundreds of processors.
- 3) They are applicable not only to joins but also to other relational query operators.

### 3.1.1 Map

A map is similar to a database scan. We define the map primitive as follows:

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

We use multiple thread groups to implement the map. Each thread group is responsible for a segment of the relation.

### 3.1.2 Scatter and Gather

We adopt the definitions of scatter and gather used by He et al. [19]. A scatter performs indexed writes to a relation, for example, hashing. Its definition is as follows, where the array  $L$  defines the distinct write location for each  $R_{in}$  tuple.

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

The 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. Its definition is as follows, where the array  $L$  defines the read location for each  $R_{in}$  tuple.

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

In general, tuples in the output relation can be a superset or a subset of the input relation in gather and scatter. For simplicity, our definitions assume the tuples in the output relation are the same set as those in the input relation.

We implemented the scatter and the gather using the multi-pass optimization scheme proposed by He et al. [19].

### 3.1.3 Prefix scan

A prefix scan applies a binary operator to the input relation of size  $n$  and generates an output relation of size  $n$  [30]. We present the definition of a prefix scan that applies the binary operator  $\oplus$  to the input relation as follows:

**Primitive:** Prefix Scan  
**Input:**  $R_{in}[1, \dots, n]$ , binary operator  $\oplus$ .  
**Output:**  $R_{out}[1, \dots, n]$ .  
**Function:**  $R_{out}[i]=\bigoplus_{j<i}R_{in}[j]$ .

An example of prefix scan is the prefix sum, which is an important operation in parallel databases [9]: Given an input relation (or array)  $R_{in}$ , the value of each output array element  $R_{out}[i]$  ( $2 \leq i \leq |R_{in}|$ ) is obtained from the sum of  $R_{in}[1], \dots$ , and  $R_{in}[i-1]$  ( $R_{out}[1]=0$ ).

We use the prefix sum implementation from the CUDA library [27]. The prefix sum has two stages, reduce and down-sweep. The reduce stage has  $\log_2|R_{in}|$  steps. In step  $i$  ( $0 \leq i < \log_2|R_{in}|$ ), thread  $j$  computes the partial sum of  $R_{in}[j*2^i]$  and  $R_{in}[(j+1)*2^i]$ . The down-

sweep stage also takes  $\log_2|R_{in}|$  steps. In step  $i$  ( $0 \leq i < \log_2|R_{in}|$ ), the partial sum is applied to  $R_{in}[j*2^i]$  and  $R_{in}[(j+1)*2^i]$ . Both stages are highly parallel on the GPU.

### 3.1.4 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. Given the partitioning fanout  $F$ , the definition of the split is as follows:

**Primitive:** Split  
**Input:**  $R_{in}[1, \dots, n]$ ,  $func(R_{in}[i]) \in [1, \dots, F]$ ,  $i=1, \dots, n$ .  
**Output:**  $R_{out}[1, \dots, n]$ .  
**Function:**  $\{R_{out}[i], i=1, \dots, n\} = \{R_{in}[i], i=1, \dots, n\}$  and  $func(R_{out}[i]) \leq func(R_{out}[j])$ ,  $\forall i, j \in [1, \dots, n], i \leq j$ .

A basic implementation is that each thread processes a portion of the input relation and inserts tuples to their target partitions. A major issue is the write conflicts among threads. They occur when multiple threads try to insert tuples into a partition concurrently. Unfortunately, there are no atomic operations such as locks for handling such conflicts on most GPUs. Thus, we propose a software approach to implement a lock-free split algorithm. The basic idea is that, prior to writing the output, we use histograms to compute the write locations of each thread. Since each thread knows its target position to write, the write conflicts among threads are avoided.

Our histogram-based algorithm is partially inspired by the parallel radix sort proposed by Zagha [39], which uses histograms to perform the radix sort. The major difference is that our histogram scheme is embedded in our primitives on the GPU. In particular, our split algorithm uses the histogram to compute the write location for each tuple (stored in the array  $L$ ) and scatters  $R_{in}$  to  $R_{out}$  according to the array  $L$ .

---

#### Algorithm 1: split ( $R_{in}, fcn, R_{out}$ )

---

**Parameters:**

$\#thread$ , the total number of threads ( $\#thread=Bp*T$ ).  
 $F$ , the partitioning fanout.  
 $tHist$ , the thread-level histogram.  $tHist[t][p]$  is the number of tuples processed by thread  $t$  and belonging to partition  $p$  ( $1 \leq p \leq F$ ).  
 $tOffset$ , the thread-level offset array.  $tOffset[t][p]$  contains the start position to output the tuples of thread  $t$  that belong to partition  $p$ .  
 $L$ , the array storing start positions to output the tuples of each partition for each thread. The start position of partition  $p$  for thread  $t$  is  $L[(p-1)*\#thread+t]$ .

- (1) Each thread constructs its  $tHist$  histogram from  $R_{in}$ .
  - (2) Each thread writes its histogram to  $L$  so that  $L[(p-1)*\#thread+t]=tHist[t][p]$ .
  - (3) Perform a prefix sum on  $L$ . The result is stored in  $L$ .
  - (4) Each thread updates its offset so that  $tOffset[t][p]=L[(p-1)*\#thread+t]$ .
  - (5) Each thread scatters its tuples to  $R_{out}$  based on its offset.
- 

Each thread group is responsible for a similar-sized portion of  $R_{in}$ . Each thread maintains a thread-level histogram ( $tHist[1 \dots F]$ ). It records the number of tuples of each partition for the thread. We use the thread-level histogram to compute the thread-level offset array ( $tOffset[1 \dots F]$ ), which contains the start position for outputting the tuples belonging to the partition for the thread.

Our split works in five steps, as illustrated in Algorithm 1. The five steps are implemented using our other primitives. The first step is implemented using a map primitive; the third one uses prefix scan; the fourth one uses a gather; and the other two use scatter.

Figure 2 shows an example of our split operation, where we divide  $R_{in}$  into two partitions. The arrows represent how the data is loaded and stored. In this example, there are two thread groups, one containing T1 and T2 and the other containing T3 and T4. The portions of  $R_{in}$  processed by different threads are in different shades. In the first step, there are four thread-level histograms. Step (2) creates the histograms and outputs them to  $L$ . Step (3) uses prefix sum to compute the offset array of each thread. For example, the start positions for writing the tuples belonging to the first and the second partitions are 0 and 4 respectively for thread T1. With these offsets, the write locations of the four threads are deterministic, and tuples can be output in parallel.

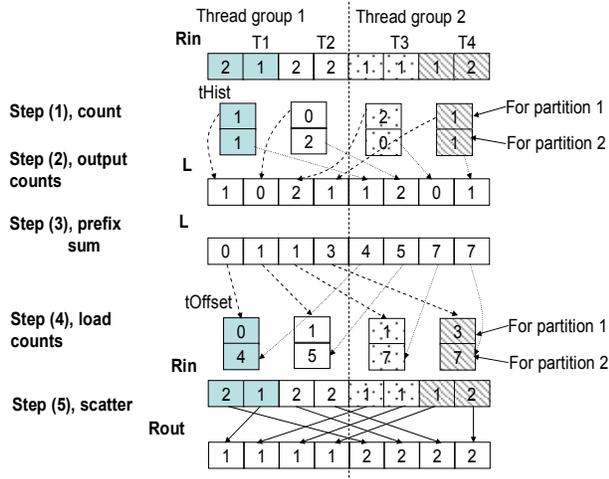


Figure 2. An example of the split primitive.

### 3.1.5 Sort

The sort primitive is used in a number of operators such as aggregation and join operators.

**Primitive:** Sort  
**Input:**  $R_{in}[1, \dots, n]$ .  
**Output:**  $R_{out}[1, \dots, n]$ .  
**Function:**  $\{R_{out}[i], i=1, \dots, n\} = \{R_{in}[i], i=1, \dots, n\}$  and  $R_{out}[i] \leq R_{out}[j], \forall i, j \in [1, \dots, n]$  and  $i \leq j$ .

We have implemented two comparison-based sorting algorithms including the bitonic sort and the quick sort. The bitonic sort uses the GPU-based bitonic sorting network [14], because independent swaps between the elements in this sorting algorithm map well to the massively threaded architecture of GPU. However, the complexity of the bitonic sort is  $O(N \log^2 N)$ , where  $N$  is the number of tuples to be sorted. In contrast, the complexity of the quick sort is  $O(N \log N)$ , which is lower than the bitonic sort. With the split primitive, the quick sort can be implemented on the GPU.

**Bitonic sort.** The bitonic sort merges *bitonic sequences* in multiple stages. A bitonic sequence is of a monotonic ascending or descending order. Given a relation  $R_{in}$ , the bitonic sorting

algorithm has  $\log_2 |R_{in}|$  stages. Stage  $x$  has  $x$  steps ( $1 \leq x \leq \log_2 |R_{in}|$ ). In Step  $i$ , it constructs bitonic sequences each of size  $2^i$ . Thus, Stage  $x$  generates the bitonic sequences each of size  $2^x$ . After  $\log_2 |R|$  stages,  $R$  is sorted. Each step of the bitonic sort performs a map on the input relation and a scatter to output the results.

**Quick sort.** The quick sort has a lower complexity than the bitonic sort. Moreover, it uses the efficient split primitive. The quick sort has two steps. First, given a set of pivots, we use the split primitive to divide the relation into multiple chunks. The pivots are chosen randomly [12]. The split process goes recursively until each chunk is smaller than a preset threshold for the chunk size. (We discuss this preset threshold in Section 3.2.2.) After the split process, we use the bitonic sort on each chunk. We choose the bitonic sort other than the insertion sort, because the bitonic sort can work entirely in the local memory and its computation maps well to the parallel execution of the GPU. We present the local memory optimization in Section 3.2.

## 3.2 Memory Optimizations

Our primitives are developed based on the many-core architecture model. The thread-level parallelism reduces the memory stalls in these primitives. However, thread-level parallelism may not completely hide the memory stalls for database workloads [17]. Thus, we utilize two memory optimization techniques on the GPU to further reduce the memory stalls: coalesced access to improve spatial locality and local memory optimization to improve temporal locality. Frequently accessed data are stored in the local memory to reduce the accesses to the device memory.

### 3.2.1 Coalesced Access

We use the map primitive to illustrate how we take advantage of the coalesced access.

The coalesced access improves the memory bandwidth utilization. Figure 3 illustrates two map schemes with and without coalesced accesses. Suppose a thread group consists of three threads. In Figure 3 (a), due to the SIMD nature of GPUs, the accesses to the device memory among the threads are consecutive during the execution. Every three concurrent accesses are coalesced into a single read. In Figure 3 (b), the accesses among threads are not consecutive. Each thread issues a distinct memory request. This results in low utilization of the memory bandwidth. Suppose every  $k$  memory requests are merged into a single request, the number of memory requests with the coalesced access is  $(k-1)$  times less than that without the coalesced access.

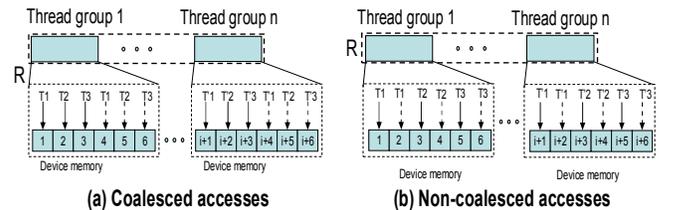


Figure 3. Maps with and without coalesced accesses.

With this optimized map primitive, the memory performance of the bitonic sort is greatly improved. Similarly, the memory accesses in steps (2) and (4) of the split primitive are also designed as coalesced ones.

### 3.2.2 Local Memory Optimization

We use the split and the sort algorithms including the bitonic sort and the quick sort as examples to illustrate the local memory optimization.

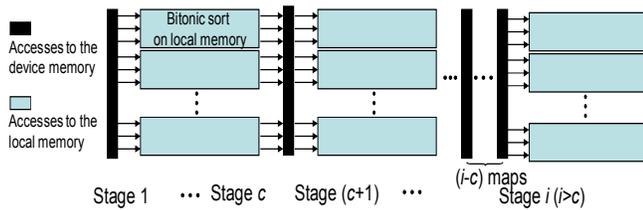
In the split, each tuple accesses the histogram and the offset array. We store these arrays in the local memory. Due to the limited size of the local memory, we determine the maximum partitioning fanout for the split. Suppose the number of partitions is  $f$  and each element in the array is encoded in  $z$  bytes. We determine  $F$  to be the maximum  $f$  such that  $T \cdot f \cdot z \leq M$ . To divide a relation into an arbitrary number of partitions,  $x$ , we apply the split operation recursively. The number of levels in the recursion is  $\lceil \log_{F^x} \rceil$ , and we uniformly set the number of partitions generated in each level of recursion.

The bitonic sort has repetitive fetches on the device memory. We propose two optimization techniques on the local memory to improve its temporal locality. These two optimizations are illustrated in Figure 4. The first optimization is applied to the first  $c$  stages ( $c = \log_2 \frac{M}{r}$ ) of the bitonic sort, which are independently performed on individual chunks of size  $M$ . We use local memory to store this chunk of data and process Stages 1 to  $c$  in the local memory. This saves  $OPT_1 = 2 + \dots + c = \frac{1}{2} c \cdot (c + 1) - 1$  fetches from the relation in the device memory. The second optimization is applied to Steps  $c, c-1, \dots$ , and 1 of Stage  $i$  ( $i > c, c = \log_2 \frac{M}{r}$ ).

These steps sort a bitonic sequence of size  $M$ . We store this bitonic sequence into the local memory at the  $(i-c)$ th step so that Steps  $c, c-1, \dots$ , and 1 process the data in the local memory. This saves  $(c-1)$  fetches in Stage  $i$  ( $i > c$ ). In total, it saves

$OPT_2 = (c-1) \cdot (\log_2 |R_{in}| - c)$  fetches for the entire bitonic sort.

Note, without the local memory optimization, the total number of times fetching the relation is  $\frac{1}{2} \log_2 |R_{in}| \cdot (\log_2 |R_{in}| + 1)$ . Suppose the relation has 16 million tuples and the local memory can hold 1024 tuples, the local memory optimization reduces the total number of times fetching the relation from originally 300 to 120.



**Figure 4. Data accesses in the bitonic sort on the GPU with local memory optimization.**

In the quick sort, since we use the bitonic sort to sort each chunk after the partitioning step, the preset threshold for the chunk size is the local memory size. Since each chunk is smaller than the local memory, the bitonic sort performs completely within the local memory.

## 4. JOIN ALGORITHMS

We now briefly describe our join algorithms, including the non-indexed and indexed nested-loop join (NINLJ and INLJ respectively), the sort-merge join (SMJ) and the hash join (HJ).

Since the GPU-based algorithms are similar to their CPU-based counterparts, we focus on the differences in our GPU-based implementations, especially their usage of our primitives. Specifically, NINLJ uses the map primitive on both relations; INLJ uses the map primitive on the outer relation; SMJ uses the sort on both relations and then maps the sorted relation for merging; HJ uses the split primitive on both relations. The result output of each join algorithm uses the prefix scan and the scatter primitives.

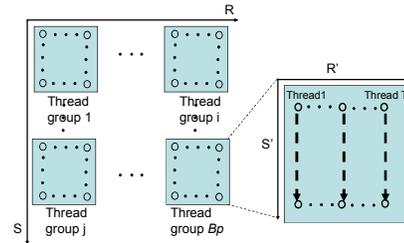
### 4.1 Join Processing

We describe the join processing of each join algorithm. Since the scheme for outputting the join result is the same for the four join algorithms, we present the result output in Section 4.2.

**Non-indexed NLJs (NINLJ).** Our algorithm is blocked nested-loops. The nested-loop join can be naturally mapped to our GPU model, as shown in Figure 5. The circles represent tuples generated by the join, some of which may be eliminated by the join predicate.

Each thread group computes the join on a portion of  $R$  and  $S$ , denoted as  $R'$  and  $S'$ , respectively. Within a thread group, each thread processes the join on one tuple from  $R'$  and all tuples from  $S'$ . The joins of the tuple from  $R'$  and other tuples from  $S$  are computed in other thread groups. Thus, the number of threads in each thread group is equal to the number of tuples in  $R'$  ( $|R'|=T$ ).

Within the join of  $R'$  and  $S'$ , we store  $S'$  into the local memory to avoid reading  $S'$  repeatedly from the device memory. Due to the limited size of the local memory, the size of  $S'$  is set to the local memory size. Since each thread group requires to access  $R'$  and  $S'$  only once from the device memory, the total volume of data transfer between the GPU and the device memory is  $\frac{|R'| \cdot |S'|}{|R'| \cdot |S'|} (|R'| + |S'|) = \frac{|R'| \cdot |S'|}{T \cdot M} (T \cdot r \cdot s + M \cdot s)$ .



**Figure 5. The non-indexed NLJ algorithm on the GPU.**

**Indexed NLJs (INLJ).** We implement the indexed join algorithm through adapting the cache-optimized search tree, CSS-Tree [31], to the GPU. Different from traditional B+-Trees that use discrete memory pointers for tree traversal, CSS-trees store the entire tree in an array and tree traversal 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.

A CSS-Tree can be efficiently constructed on the GPU taking a sorted relation as input. In the presence of the tree index, the indexed join consists of two major steps, searching for the first occurrence of matching tuples in the indexed relation, and then accessing the indexed relation for join results. While searching for a single key in such a tree offers little opportunity for parallel processing, multiple searches, however, fit extremely well into the

parallel programming model. Therefore, multiple keys are searched in parallel on the tree. Given a relation  $R$ , the search starts at the root node and steps one level down the tree in each iteration until reaching the data nodes on the bottom. A binary search or a sequential search is used to locate the index of the node to go. The binary search has fewer comparisons but has more branch divergence among the threads than the sequential search. We empirically evaluated these two search methods in Section 5.4.

Since the upper levels of the tree index are frequently accessed, we replicate the upper levels of the tree index to the local memory. Given the tree fanout  $f$ , and tree node size  $z$ , the total size of tree nodes in the upper  $l$  levels is  $z \frac{f^l - 1}{f - 1}$ . We compute the number of levels that can be replicated into the local memory as the maximum  $l$  such that  $z \frac{f^l - 1}{f - 1} \leq M$ .

**Sort-Merge Joins (SMJ).** Similar to the traditional sort-merge joins, we first sort the two relations and then perform a merge step on these two sorted relations.

The merge step is done in parallel to fully utilize the computation resources. The basic idea is to perform the merge on a chunk of  $S$  and its matching chunk of  $R$  independently. 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. Each thread group is responsible for a pair. In the merge process, the  $S$  chunk is first loaded into the local memory. Next, the  $R$  tuples are used to find the matching results. Each thread reads a tuple from  $R$  and performs a search on the  $S$  chunk for matching. Either a sequential search or a binary search can be used.

**Hash joins (HJ).** We develop a parallel version of the radix hash join [10]. The radix partitioning is implemented using our split primitive. Our algorithm has two phases.

Phase 1) Partitioning. We split  $R$  and  $S$  into the same number of partitions using  $\log_2(\|S\| / M)$  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.

Phase 2) Matching. We choose the smaller one of the  $R$  and  $S$  partitions as the inner partition to be loaded into the local memory, and the other one as the outer partition. Each tuple from the outer partition uses a sequential search or a binary search on the inner partition for matching. If the binary search is used, we use the bitonic sort to sort the inner partition prior to probing.

## 4.2 A Lock-Free Scheme for Result Output

Since the GPU lacks incremental memory allocation on the device memory during the kernel execution, the result output has two major problems. The first one is the unknown join result size. One may consider estimating the (maximum) number of results for each thread. However, since the maximum number of results for

the join of  $m$  by  $n$  tuples is  $m \times n$ , this upper bound usually exceeds the size of the device memory. The second one is that write conflicts occur when multiple threads write results to the shared output region in the device memory. We propose a three-phase scheme to solve these two problems.

First, each thread counts the number of join results for the partitioned join it is responsible for. A counter is maintained locally. There is no conflict in this step, because no threads write the actual join result.

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. Through the prefix sum, we also know the total number of results generated by the join.

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. Since each thread has its deterministic positions to write to, any write conflicts are avoided. If the size of the join result is larger than the device memory, we output the join results in multiple passes. In each pass, we output the join results from a portion of the threads.

This three-phase scheme does not require the hardware support of atomic functions. However, it requires evaluating the join predicates twice. Fortunately, with the GPU's high computation power, the extra join predicate evaluation poses little overhead.

## 4.3 Skew Handling

In the partitioning-based algorithms such as SMJ and HJ, the skew in the data results in an imbalanced partition size. The processing of an inner partition that is larger than the local memory requires accesses to the device memory. Consequently, it may suffer from the memory stall and hurt the overall performance.

The first problem is to identify the partitions that do not fit into the local memory. Taking the input array of partition sizes (i.e., the element  $i$  in the array is the size of the  $i$ th partition), we use the split primitive to divide the partitions into two groups, one for the partitions larger than the local memory and the other for those not larger than the local memory.

Once we identify the partitions that are larger than  $M$ , we further decompose each of these partitions into multiple chunks each of size  $M$ , and process these generated small chunks in the local memory. For the SMJ, we perform a merge step on all possible matching pairs of chunks. For the HJ, we use the NINLJ on each matching pair of chunks.

## 5. EXPERIMENTS

In this section, we evaluate the performance of our proposed GPU primitives and join algorithms in comparison with the algorithms on the CPU.

### 5.1 Experimental Setup

We have implemented and tested our algorithms on a PC with a NVIDIA 8800 GTX GPU and a recently-released Intel Core2 Duo Quad-Core processor. The hardware configuration of the PC is shown in Table 2. 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. Thus, GPU and CPU have a theoretical bandwidth 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.

**Table 2. 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

We used synthetic data sets and workloads for detailed studies on our join algorithms. Our homegrown workload contains two join queries on relations  $R$  and  $S$ . Relations  $R$  and  $S$  are binary tables each consisting of two four-byte integer attributes, the record ID ( $rid$ ) and the key value ( $key$ ). We used both uniform and non-uniform key values. We generated our non-uniform key values by setting a certain percentage of tuples to be a constant key value (e.g., one in our experiments). Other tuples are randomly distributed. When this percentage is zero, key values in the relation are uniformly distributed; when it is 100%, all tuples have the same key value. We varied this percentage to simulate different degrees of skewness.

The join queries in our own workloads are “SELECT  $R.rid, S.rid$  FROM  $R, S$  WHERE <predicate>”. We used an equijoin and a non-equijoin query. The equi-join takes  $R.key = S.key$  as the predicate and the non-equijoin  $R.key \leq S.key \leq R.key + \delta$  ( $\delta$  is a constant integer).

Considering different parameters in our workload, we performed three sets of experiments on the equijoin query. First, we fixed the size of  $R$  and varied the size of  $S$ . The key values of  $R$  and  $S$  are uniformly distributed. For NINLJ, we fixed  $|R|$  to be 1 million; while for the other three joins, we fixed  $|R|$  to be 16 million. Second, we examined the performance impact of varying the join selectivity. Third, we investigated our algorithms on the non-uniform data sets. In the later two sets of experiments, we fixed both  $|R|$  and  $|S|$  to be one million for NINLJ. For the other three joins, we fixed both  $|R|$  and  $|S|$  to be 16 million. This is our *default* experimental setting for data sizes unless specified otherwise. These settings were chosen to be similar to the previous studies on in-memory join algorithms [10]. Finally, we varied the  $\delta$  value in the non-equijoin predicate and examined the performance of non-equijoins.

In addition to supporting the regular data types such as integers, our primitives and join algorithms support more complex data types such as strings. We support more complex data types through indirection by storing offsets and lengths in our record. Specifically, the values of the field of all tuples are consecutively stored into an array named data array. We represent the field of each tuple using a pair ( $offset, length$ ), where  $offset$  is the start position of the value in the data array and  $length$  is the length of the value (in bytes). The value is fetched according to the offset

and the length. If two tuples need to be swapped, we swap them without modifying the data array.

We run each experiment ten times and report the average value.

**Implementation details on CPU.** For comparison, we have implemented highly optimized CPU-based primitives and join algorithms. The primitives are designed to be parallel and run on the quad-core machine. We use cache optimization techniques [34] to fine tune the performance of the parallel implementation. With these optimized primitives, we implement four join algorithms including the blocked NINLJ [34], the INLJ with the CSS-tree index [31], the SMJ with the optimized quick sort [22] and the radix HJ [10]. We compiled our algorithms using MSVC 8.0 with full optimizations. Moreover, we used OpenMP [29] to implement the threading mechanism on the CPU. In general, the parallel CPU-based primitives and join algorithms are 2-6X faster than the sequential ones on the quad-core CPU. To check whether our CPU-based implementation has a comparable performance with state-of-the-art main memory databases, we also performed a performance comparison between our algorithms and MonetDB [26]. The comparison was done on the core query processing algorithms, excluding the other components such as query parsing and plan generation.

**Implementation details on GPU.** We implemented our primitives and join algorithms using CUDA [27] and DirectX10 [6]. DirectX is a common graphics API runnable on most GPUs including AMD’s and NVIDIA’s. In contrast, 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++. Since the performance results of the DirectX implementation are similar to those of CUDA, we discuss our implementation and results of CUDA in detail and briefly present the results for DirectX.

Our GPU-based joins can be easily mapped to the CUDA framework. To implement an efficient algorithm using CUDA, we need to determine the following parameters with respect to the target operation: the number of threads for each thread group ( $T$ ) and the number of thread groups ( $Bp$ ). Issuing more threads to the GPU can potentially improve the overall performance by hiding memory latency at the cost increasing register pressure. Moreover, due to the limited local memory space on the multiprocessors,  $Bp$  and  $T$  cannot be arbitrarily large. Through experiments, we find that  $Bp=128$  and  $T=64$  are a good tradeoff value, where the memory latency is sufficiently masked, and each block receives adequate computation resources. Taking the search performance and data transfer into account, we set the number of keys in a node of the GPU CSS-tree to be 32.

The DirectX10 programmable pipeline contains multiple stages including the Vertex Shader (VS), Geometry Shader (GS) and Pixel Shader (PS). For each algorithm, we draw a set of points corresponding to a tuple processing. The map, the gather and the scatter are implemented as vertex texture fetches and positioning in the VS. These APIs have inherent thread parallelism and achieve a similar performance to the CUDA implementation. The prefix scan is adopted from Horn’s algorithm [20] and our sorting

algorithm is the bitonic sort [14]. We implement the split primitive as two steps, first sorting the tuples according to their partition identifiers, and next scattering these tuples with Min and Max blending to obtain the start and the end positions of each partition. The matching process is performed by drawing points from the outer relation, each point probing the inner relation. Since the local memory is not exposed to DirectX, NINLJ stores the inner block in the constant buffer, a fast on-chip cache exposed to DirectX. INLJ probes the texture storing the CSS-tree using the search keys of the outer relation. The fanout of the CSS-tree is set to four, which is the number of color channels on the GPU for parallel comparison. SMJ sorts the texture of the inner relation and performs binary search for matching results. HJ builds the hash table for the inner relation in the texture using the split and renders the outer relation to probe the textures of the hash table. Unlike the three-phase result output scheme in the CUDA implementation, the DirectX implementation utilizes the stream-out feature of the GS to output the join results in parallel.

## 5.2 Data Transfer between Device Memory and Main Memory

Figure 6 shows the memory copy time 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, the copy time increases as the block size decreases. When the block size is larger than 4MB, the copy time remains almost constant. That means when the relation size is larger than 4MB, the bandwidth is fully utilized. These results suggest that the programmer could batch small data transfers to reduce the time of data transfer between the GPU and the CPU.

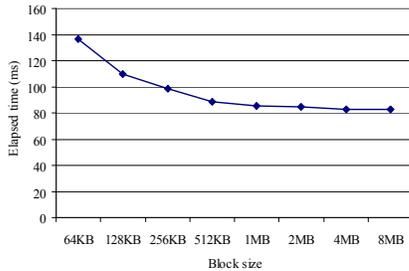


Figure 6. Data transfer time from the main memory to the device memory ( $||R||=256\text{MB}$ ). When the block size is larger than 4MB, the peak bandwidth is 3.1 GB/sec.

## 5.3 Results on Primitives

Since GPU-based primitives are usually used as intermediate components in the GPU-based join algorithms, their input data are already in the device memory and their output data are stored in the device memory as input to other primitives. Thus, we exclude the time of data transfer between the GPU and the CPU in the results for the primitives on the GPU.

Table 3 shows the elapsed time of optimized primitives when  $|R|$  is fixed to be 16 million. The locations in the scatter and the gather are random. For the prefix scan, we compute the prefix sum on 16 million integers. The split function for the split is  $fcn(x) = x$

$\text{mod } 64$ . That means the split divides the relation into 64 partitions.

We define the *speedup* to be the ratio of the execution time on the CPU to that on the GPU. Overall, the GPU-based primitives achieve a performance speedup of 2-27X over the CPU-based primitives. We obtained similar performance speedup with the data size varied. This speedup is due to the high parallelism and the two memory optimizations.

Table 3. Elapsed time for primitives ( $|R|$  is 16 million). The speedup of the GPU-based primitives is 2-27X over the CPU-based primitives.

Primitive	CPU (ms)	GPU (ms)	Speedup
Map	109	4	27.3
Scatter	1312	104	12.6
Gather	1000	103	9.7
Prefix scan	141	14	10.1
Split	813	125	6.5
Sort(qsort)	2313	945	2.4

We have the following four observations. First, the average bandwidth of the optimized map primitive is 2.4GB/sec and 64GB/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 86GB/sec. Second, the scatter and the gather have a much lower bandwidth 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. Forth, 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 results is not shown in the table). This result is consistent with the fact that the quick sort has lower complexity than the bitonic sort. We used the quick sort as our sorting primitive in the CUDA implementation.

The speedups on the scatter, gather and prefix scan primitives are similar to those in the previous work [19][33]. Thus, we discuss the map, split and sort primitives in more detail. Specifically, we studied the performance impact of the three optimizations on the GPU.

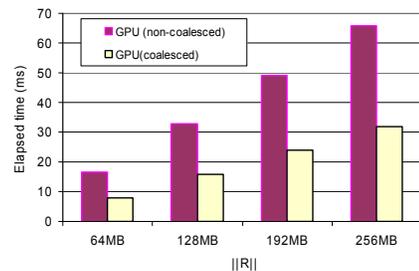


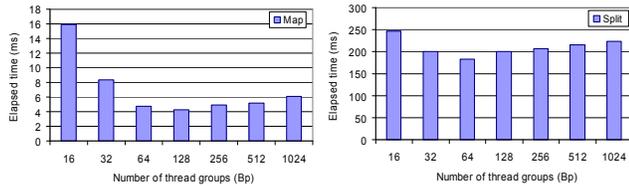
Figure 7. The map performance with and without the coalesced access. The coalesced access improves the map bandwidth on the GPU by around twice.

**Coalesced access.** Figure 7 shows the performance of the map primitive with the relation size varied. To isolate the performance impact of the coalesced access from the thread parallelism, we set  $Bp=16$  and  $T=32$ , which is equal to the number of multiprocessors and the number of threads in a schedule unit of G80, respectively. The coalesced access improves the map bandwidth on the GPU by

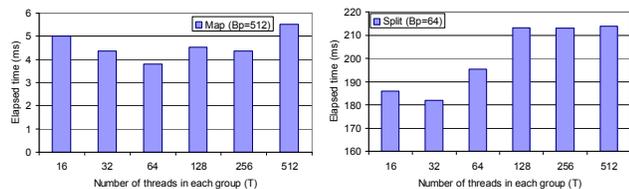
a factor of about two. Note, the bandwidth of the coalesced map is 4.5GB/sec, which is far lower than the theoretical bandwidth due to the absence of high parallelism and the local memory optimization.

**Thread parallelism.** Figure 8 shows the elapsed time with a varying number of thread groups for the map and the split primitives. The number of threads in each thread group is fixed to be 32. Since the results for the bitonic sort are similar to those of the map, and the results for the quick sort are similar to the split, the results for the sort are omitted. The map primitive is implemented with coalesced accesses. As  $Bp$  is smaller than a threshold value, the elapsed time of both algorithms greatly decreases as the  $Bp$  value increases. This is because the memory stalls of accessing the device memory are better hidden by computation and the increase in bandwidth utilization. Since the map is cheaper than the split, the performance impact of increasing the number of thread groups is more significant on the map than on the split. When  $Bp$  is larger than the threshold value, the elapsed time slightly increases as  $Bp$  increases. The suitable numbers of thread groups are 128 and 64 for the map and the split, respectively.

After obtaining the suitable number of thread groups, we further varied the number of threads per thread group. The results are shown in Figure 9. When  $T$  is smaller than a threshold value, the elapsed time decreases as  $T$  increases. This indicates memory stalls can be further hidden by increasing the number of threads per thread groups. When  $T$  is larger than the threshold value, the performance degrades due to the computation resource contention on the multiprocessor. The suitable numbers of threads per thread group for the map and the split are 64 and 32, respectively. Note, with the optimization of the coalesced access and the thread parallelism, the map primitive achieves a bandwidth of 64 GB/sec.



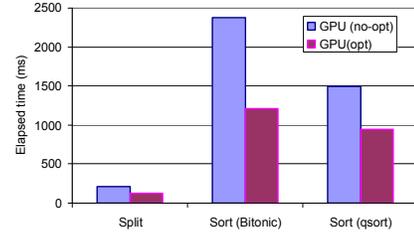
**Figure 8. The elapsed time with the number of thread groups varied. The number of threads in each thread group is fixed to 32. The best numbers of thread groups in the map and the split are 128 and 64, respectively.**



**Figure 9. The elapsed time with the number of threads per thread group varied. The number of thread groups is set to be the best one shown in Figure 8. The suitable numbers of threads per thread group for the map and the split are 64 and 32, respectively.**

**Local memory optimization.** Figure 10 compares the GPU-based primitives with and without the local memory optimization. The local memory optimization improves the overall performance of

the split and the sort primitives by 1.5-2X. This indicates that the thread parallelism may not fully eliminate the memory stalls when accessing the device memory. With the local memory optimization, the efficiency of the GPU-based primitives is greatly improved.

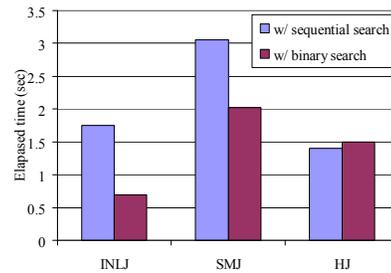


**Figure 10. The performance impact of the local memory optimization. The shared memory optimization improves the overall performance by 1.5-2X.**

### 5.4 Results on Joins

Since searching the data in the local memory is a core operation in the join step, we first studied the binary search and the sequential search. Figure 11 compares the performance of our joins with binary search and with sequential search. The result for NINLJ is not shown, because binary search is not used in NINLJ. The result for INLJ does not include the time for constructing the tree index. The tree construction time is so small that it can be ignorable on the GPU. Additionally, we observed that the local memory optimization achieved a performance improvement of around 10% on INLJ.

Although the sequential search takes fewer branches than the binary search, the binary search has fewer data accesses than the sequential search. The binary search improves the performance by 2.5X and 1.5X on INLJ and SMJ, respectively. The binary search has a relatively high speedup in the INLJ, because the search on the tree node is the major operation of INLJ. In contrast, the binary search degrades the performance of HJ due to the overhead of the extra sorting on the local memory.



**Figure 11. Elapsed time of join algorithms with binary search and sequential search.**

**Table 4. Elapsed time of the four relational joins on the GPU and the CPU for the uniform data sets. The speedup of the GPU-based primitives is 2-7X over the CPU-based primitives.**

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

Table 4 shows an end-to-end comparison on the elapsed time of the four relational joins on the GPU and the CPU. The elapsed time on the GPU includes the data transfer time between the device memory and the main memory. Overall, the GPU-based joins have a 2-7X speedup over the CPU-based joins. The high performance speedup is due to the efficient primitives as well as the efficient matching on the data in the local memory.

Figure 12 shows the time breakdown of the four join algorithms on the GPU. We divide the total execution time of a GPU-based join into three components including the time for copying input data into the device memory, join processing and result output to the main memory. For all join algorithms, the join processing time is dominant. The total time of copying data between the main memory and the GPU memory (one time cost for each join) was around 0.1%, 13%, 4% and 6% of the total execution time of NINLJ, INLJ, SMJ and HJ, respectively. Copying the input relations and outputting the join results are bulk transfers with a block size larger than 4MB. Thus, the bandwidth between the main memory and the device memory is fully utilized.

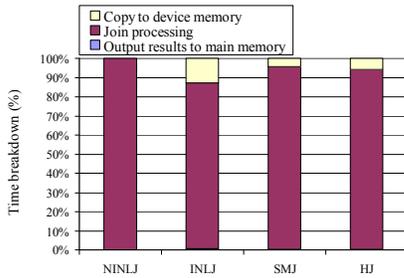


Figure 12. Join processing time and data transfer time between the main memory and the device memory.

We studied the join performance with varying workload characteristics. Figure 13 shows the speedup of the GPU-based joins over the CPU-based joins with varying join selectivity and percentage of duplicates in  $R$ . The larger the join selectivity is, the larger the join output. The speedup is stable when the join selectivity varies. This result indicates that the data transfer time between the device memory and the main memory has little performance impact on GPU-based joins. As the percentage of duplicates increases, the relation becomes more skewed. The speedup of the SMJ and HJ is stable. This indicates the effectiveness of our skew handling.

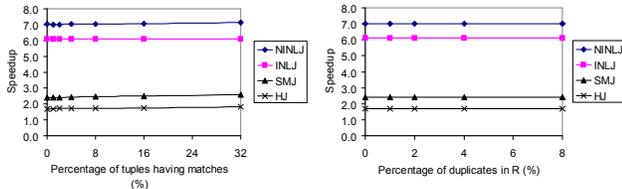


Figure 13. The speedup of the GPU-based joins over the CPU-based joins: (left) the join selectivity is varied; (right) the percentage of duplicates in  $R$  is varied and  $S$  is uniform.

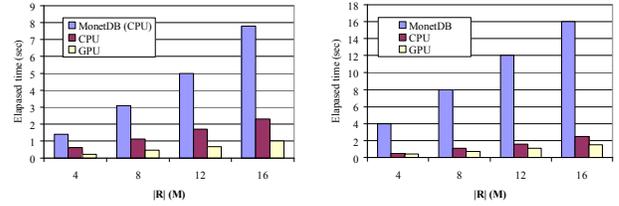


Figure 14. Performance comparison between our algorithms and MonetDB: (left) sort, (right) hash joins.

## 5.5 Comparison with MonetDB

Figure 14 compares the performance of the sort and the hash join. We varied  $|R|$  from 4M to 16M for the sort. For the hash join, we kept  $|R| = |S|$ , and varied both  $|R|$  and  $|S|$  from 4M to 16M. As the data size increases, both our CPU- and GPU-based implementations outperform MonetDB. This figure indicates that the efficiency of our implementation is comparable to MonetDB.

## 5.6 Results on DirectX Implementation

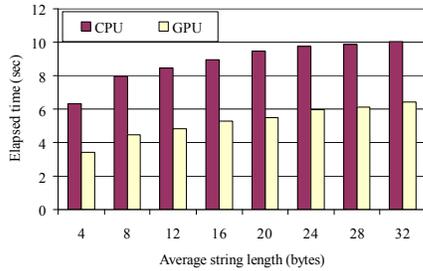
Table 5 shows the elapsed time of the four relational joins implemented with CUDA and DirectX. For each implementation, we show its total execution time and the time for the join processing only, i.e., the texture copy in/out and encoding/decoding time is not included for the DirectX measurement and the data copy in/out is not included for the CUDA measurement. The DirectX-based NINLJ and INLJ achieve a similar performance to their CUDA-based counterparts. The GPU pipelines in these DirectX implementations are short and simple. In contrast, the DirectX-based SMJ and HJ are about twice as slow as their CUDA-based counterparts. These DirectX implementations contain more graphics related overhead such as texture coding/decoding.

Table 5. Elapsed time in seconds of the four relational joins implemented with CUDA and DirectX (DX).

	DX (join)	DX (total)	CUDA (join)	CUDA (total)
NINLJ	72.3	74.1	75.0	75.0
INLJ	0.7	0.9	0.6	0.7
SMJ	3.8	4.7	1.9	2.0
HJ	2.3	2.7	1.2	1.3

## 5.7 Handling Other Data Types

Figure 15 shows the performance of our sort primitive on the strings. Each tuple contains two fields, the record ID and the string field. The number of tuples is fixed to be four million, and we varied the average string length. As the string length increases, the variance of the string lengths increases. The performance speedup of our GPU-based sort over the CPU-based sort slightly decreases. One possible reason is the increasing branches in the string matching. Nevertheless, the speedup of the GPU-based quick sort is around 2X over its CPU-based counterpart.



**Figure 15. Sorting strings on the CPU and the GPU. The GPU-based quick sort achieves a speedup of around 2X over its CPU-based counterpart.**

## 5.8 Summary

In summary, our GPU-based primitives and join algorithms achieve a speedup of 2-27X over their optimized CPU-based counterparts. We evaluated our join algorithms for both equijoins and non-equijoins, different data sizes, join selectivities and data distributions. Generally, INLJ is the suitable join algorithm in the presence of the index, and NINLJ for non-equijoins and HJ for equi-joins otherwise. The performance speedup for the non-indexed NLJ, the indexed NLJ, the sort-merge join and the hash join is over 7.0X, 6.1X, 2.4X and 1.9X, respectively.

## 6. DISCUSSION

We first discuss the performance speedups, and next the opportunities and the limitations of query processing on the GPU.

The performance speedup of our GPU-based join algorithms over quad-core CPU-based join algorithms is resulted from the differences in the architectures as well as the algorithm design. First, the G80 has 18X more total clock cycles and over 12X higher memory bandwidth than the quad-core CPU. The speedups of our join algorithms are smaller than both ratios, mainly due to the inter-thread communication on the GPU. Second, the L2 cache of the quad-core CPU is 32X larger than the local memory on the GPU. Since memory stalls are a significant performance factor, memory optimizations are important in the algorithm design. On the GPU, we utilize the coalesced access to improve the bandwidth utilization, and the local memory optimization for the temporal locality. In comparison, our CPU-based algorithms have only cache optimization for temporal locality. It would be interesting to quantitatively study the performance impact of each individual hardware feature.

Through designing and implementing relational join algorithms on GPUs, we have identified a number of opportunities and limitations of new-generation GPUs as a database query co-processor.

The following are four major opportunities:

First, GPUs have a highly parallel hardware architecture that fits extremely well with data-parallel query processing. The massive thread parallelism of the GPU hides the memory latency more efficiently than traditional von Neumann architectures. Moreover, the high memory bandwidth and the fast inter-processor communication can significantly accelerate the performance of many database operations.

Second, the GPU programmability for general-purpose computing has been improving greatly. The AMD CTM and NVIDIA CUDA

APIs extend the functionality of GPUs for the high-performance computing market in addition to the traditional gaming market.

Third, with the new architectural features and the improved general-purpose programmability, new-generation GPUs allow us to utilize traditional wisdom from both the GPU programming model and the CPU-based query processing techniques. Specifically, we adapt CPU-based optimization techniques to the GPU hardware features in order to reduce memory stalls of the primitives and the join algorithms on the GPU.

Fourth, our primitive-based methodology has a high flexibility for the computation on many-core architectures including GPUs and multi-core CPUs. We proposed to break the four basic join algorithms into a set of simple primitives. The algorithms of these primitives are scalable to hundreds of processors. Moreover, these primitives can be used to develop higher-level primitives and other applications. Additionally, we can easily replace the existing implementation of a certain primitive with a more efficient one whenever applicable. For instance, GPGPU researchers recently released CUDPP [18], a CUDA library of data parallel primitives. We plan to compare it with our own primitives, and choose the more efficient ones to implement the join algorithms.

We also identified a few limitations of GPUs for performing relational query processing:

First, query processing in general and join processing in specific is a complex task in its runtime logic in addition to its data-intensiveness. Mapping such a task onto the SIMD processors in the GPU requires a significant amount of design and implementation effort. In particular, the SIMD architecture by design trades functional simplicity for high efficiency and concurrency. For instance, branches frequently appear in query processing algorithms, e.g., index searches, and need special care on the GPU. Existing techniques [40] for rewriting the branches on the CPU can also be applied to the GPU. This rewrite is especially useful for common and expensive operations. We acknowledge that this kind of rewriting in general is a difficult task for the run-time environment.

Another example is that the synchronization mechanism for handling read/write conflicts, which happen constantly in query processing, is limited in the GPU. As a result, our primitives and join algorithms take extra computation such as computing the writing offsets to avoid the conflicts. This extra computation increases the work complexity of our algorithms by a constant factor.

Second, with the exposure of the massively multi-threaded hardware architecture on the GPU, it also makes GPGPU programming trickier to ensure correctness and to fully utilize the essential GPU features such as data parallelism than the previous GPUs. In our work, we have developed a small set of primitives that are carefully designed and highly tuned for GPU join processing. Similarly, GPGPU programmers could produce better and faster programs using a set of well-defined primitives as building blocks to address this issue.

Third, even though the latest GPU frameworks, such as CTM and CUDA, are a significant leap from the traditional GPUs in providing great details about the hardware architecture, they are still far behind the CPU vendors' tradition of giving sufficient details about the hardware specification, e.g., the memory hierarchy. Currently, we mainly rely on empirical experiments to

estimate the hardware parameters and to identify the suitable settings for our algorithms.

Fourth, the power consumption of the GPU is higher than that of the CPU. In our experiments, the GPU requires a power supply of 450 Watts, whereas the CPU requires 95 Watts only. It is desirable to develop software or hardware techniques to reduce the power consumption of the GPU.

Finally, as a co-processor, the GPU requires advanced software techniques to support complex workloads. For example, lacking hardware support for complex data types is an inherent weakness of the GPU. Currently, we can use software solutions for supporting more complex data types such as high precision numbers on the GPU [38]. Fortunately, GPU vendors plan to support high precision numbers such as double in the near future.

## 7. 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 even more suitable for database query processing than before. In this study, we have designed a small set of data-parallel primitives for relational join processing on GPUs. These primitives provide high-level abstractions for data-centric operations and are highly tuned to fully utilize the architectural features of graphics processors. We have implemented four representative relational join algorithms using these primitives and have compared the join performance with optimized CPU-based in-memory join algorithms. We find that our GPU joins achieve a speedup of 2-7X over their optimized CPU-based counterparts.

This paper focuses on GPU join processing in the video memory. We believe this is an important but initial step towards building a high-performance, general-purpose database query processor using the GPU. One interesting future direction is to evaluate our join algorithms with more complex workloads. Additionally, we are interested in how to schedule the execution of relational query processing between the GPU and the CPU so that their computation power is fully exploited.

## 8. Experiment Repeatability

The repeatability committee has not been able to repeat the experiments of this paper due to the lack of appropriate hardware. Code and data used in the paper are available at <http://www.sigmod.org/codearchive/sigmod2008/>.

## 9. Acknowledgement

The work of Ke Yang was done while he was visiting HKUST, and the work of Rui Fang was done when he was an undergraduate student at HKUST. This work was supported by grant 617206 from the Hong Kong Research Grants Council.

## 10. REFERENCES

- [1] A. Ailamaki, N. Govindaraju, S. Harizopoulos and D. Manocha. Query co-processing on commodity processors. VLDB, 2006.
- [2] AMD CTM, <http://ati.amd.com/products/streamprocessor/>.
- [3] S. Azadegan, A. R. Tripathi. Parallel join algorithms for SIMD models. ICPP (3) 1991: 125-133.
- [4] S. Azadegan, A. R. Tripathi. A parallel join algorithm for SIMD architectures. Journal of Systems and Software 39(3), 1997.
- [5] N. Bandi, C. Sun, D. Agrawal and A. El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. VLDB, 2004.
- [6] D. Blythe. The Direct3D 10 system. SIGGRAPH 2006.
- [7] C. Boyd. Mass market applications of massively parallel computing. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2007.
- [8] I. Buck, T. Foley, D. Horn, J. Sugerma, K. Fatahalian, M. Houston and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. SIGGRAPH 2004.
- [9] G. E. Blelloch. Prefix sums and their applications. Technical report, CMU-CS-90-190, Nov 1990.
- [10] P. Boncz, S. Manegold and M. Kersten. Database architecture optimized for the new bottleneck: memory access. VLDB, 1999.
- [11] J. Cieslewicz, J. Berry, B. Hendrickson and K. A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. DaMoN, 2006.
- [12] T. Cormen, C. Leiserson, R. Rivest and C. Stein. Introduction to Algorithms, Second Edition. 2001.
- [13] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. In Communications of the ACM, Vol. 35, No. 6, June 1992.
- [14] N. Govindaraju, J. Gray, R. Kumar and D. Manocha. GPU TeraSort: high performance graphics coprocessor sorting for large database management. SIGMOD, 2006.
- [15] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. SIGMOD, 2004.
- [16] N. Govindaraju, N. Raghuvanshi and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. SIGMOD, 2005.
- [17] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: limitations and opportunities. CIDR, 2007.
- [18] M. Harris, J. Owens, S. Sengupta, Y. Zhang and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. <http://www.gpgpu.org/developer/cudpp/>, 2007.
- [19] B. He, N. Govindaraju, Q. Luo and B. Smith. Efficient gather and scatter operations on graphics processors. ACM/IEEE Supercomputing, 2007.
- [20] D. Horn. Stream reduction operations for GPGPU applications. In GPU Gems 2, Ed. Addison Wesley, 2005.
- [21] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. VLDB, 1991.
- [22] A. Lamarca and R. Ladner. The influence of caches on the performance of sorting. SODA, 1997.
- [23] M. D. Lieberman, J. Sankaranarayanan, H. Samet. A fast similarity join algorithm using graphics processing units. ICDE, 2008.
- [24] B. Liu and E. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. VLDB, 2005.
- [25] H. Lu, K. Tan and M. Shan. Hash-based join algorithms for multiprocessor computers with shared memory. VLDB, 1990.
- [26] MonetDB. <http://monetdb.cwi.nl/>.

- [27] NVIDIA CUDA (Compute Unified Device Architecture), <http://developer.nvidia.com/object/cuda.html>.
- [28] OpenGL, <http://www.opengl.org/>.
- [29] OpenMP, <http://www.openmp.org/>.
- [30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* (26), 2007.
- [31] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. *VLDB*, 1999.
- [32] D. A. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD*, 1989.
- [33] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens. Scan primitives for GPU computing. *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2007.
- [34] A. Shatdal, C. Kant and J. F. Naughton. Cache conscious algorithms for relational query processing. *VLDB*, 1994.
- [35] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik. *C-Store: A column-oriented DBMS*. *VLDB*, 2005.
- [36] C. Sun, D. Agrawal and A. El Abbadi. Hardware acceleration for spatial selections and joins. *SIGMOD*, 2003.
- [37] D. Tarditi, S. Puri and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *ASPLOS*, 2006.
- [38] A. Thall. Extended-precision floating-point numbers for GPU computation. *SIGGRAPH*, 2006.
- [39] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. *ACM/IEEE Supercomputing*, 1991.
- [40] M. Zukowski, S. Heman, N. Nes, P. Boncz. Super-Scalar RAM-CPU Cache Compression. *ICDE*, 2006.