Relational Joins on Graphics Processors

Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju*, Qiong Luo, and Pedro Sander

Hong Kong Univ. of Science and Technology, China *Microsoft Corporation, USA

{saven, keyang, rayfang, mianlu, luo, psander}@cse.ust.hk nagag@microsoft.com

ABSTRACT

We present our novel design and implementation of relational join algorithms for new-generation graphics processing units (GPUs). The new features of such GPUs include support for writes to random memory locations, efficient inter-processor communication through fast shared memory, and a programming model for general-purpose computing. Taking advantage of these new features, we design a set of data-parallel primitives such as scan, scatter and split, 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 to effectively hide the memory latency. We have implemented our algorithms on a PC with an NVIDIA G80 GPU and an Intel P4 dual-core CPU. Our GPU-based algorithms are able to achieve 2-20 times higher performance than their CPU-based counterparts.

1. INTRODUCTION

Modern graphics processing units (GPUs) are highly specialized commodity architectures designed for gaming applications. GPUs can be regarded as massively parallel processors with 10x faster computation and 10x higher memory bandwidth than CPUs [3]. For instance, the current NVIDIA GeForce 8800 GTX (G80) card costing $550 has over a hundred processors executing instructions in parallel and supports over one billion concurrent threads, with an observed performance of 330 GFLOPS and a peak memory bandwidth of 86 GB/seconds [2]. Moreover, GPU performance is improving at a rate that is higher than Moore's law for CPUs, and current GPUs provide general parallel programming capabilities including support for scatter operations and inter-processor communication. Furthermore, due to better programming language support on GPUs, current GPUs can be exploited for general purpose computations including database operations [4] [11][12] [13].

In this paper, we investigate the design and implementation of common relational join algorithms on GPUs. Specifically, we present algorithms for non-indexed and indexed nested-loop joins (NINLJ and INLJ respectively), the sort-merge join (SMJ) and the hash join (HJ). These CPU-based join algorithms have been studied extensively in the literature. Many variants have been designed for in-memory databases [7][18][23][25] and for parallel databases [8][9][17][20][21][24]. 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.

As the hardware architecture of GPUs differs significantly from CPUs, we explore how to map algorithms for relational joins on GPUs. For instance, current GPUs provide parallel lower-clocked execution capabilities on over a hundred processors whereas current CPUs offer out-of-order execution capabilities on smaller number of cores. Moreover, the majority of GPU transistors are devoted to computation rather than caches (or shared memory), 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 in many data-parallel algorithms but also require careful algorithmic design to achieve high performance.

GPU-based algorithms share a similar degree of complexity with prior parallel database algorithms. In particular, the SIMD design in GPUs, its massively multithreaded capability, and its high video memory latency, require our algorithms to achieve good load balance across processors to hide the latency effectively [14]. Moreover, GPUs do not provide hardware support for handling the read/write conflicts on the shared data among threads. This lack of hardware support avoids performance overhead to maintain cache coherence. Consequently, high-level abstractions and carefully designed patterns are necessary for the software correctness and efficiency.

Considering the characteristics of GPUs and individual join algorithms, we design a set of data-parallel primitives that are commonly used in the join algorithms and are highly tuned for the GPU. Most of these primitives can find their functionally-equivalent CPU-based counterparts in traditional databases, but differ significantly in design and implementation. For instance, our scan primitive employs the coalesced reads among GPU threads to fully utilize the video memory bandwidth; our split avoids the read/write conflicts by aligning histograms to the GPU threading architecture efficiently; our scatter and gather work in multiple passes for an improved spatial locality in the memory access; and our sort maps the GPU to a sorting network to provide fast and deterministic performance irrespective of data skews. Furthermore, wherever possible, our primitive algorithms take advantage of newly exposed functionality that allows inter-processor communication through shared memory.

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 block of threads within a processor; our INLJ constructs a GPU-based variant of the CSS-Tree (Cache-Sensitive Search Trees) [23] and performs a massive number of concurrent index searches in the join; our SMJ utilizes quantiles for balanced range-partitioning and merges the sorted partitions in parallel; and our HJ recursively partitions data into the shared memory and performs joins on the matching partitions in parallel. We have implemented all of our GPU-based join algorithms and evaluated them on a PC in comparison with their optimized CPU-based counterparts. All join algorithms operate on memory-resident data organized in the column-based model [7][14][26].

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. In particular, 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. Third, we discuss our lessons learned from experience and provide insights and suggestions for GPU vendors as well as the GPGPU (General-Purpose Computation on GPUs) community.

The remainder of this paper is organized as follows. In Section 2, we briefly review GPU- and CPU-based query processing techniques and parallel join algorithms. In Section 3, we present details of the latest GPU architecture, 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 discuss the related work on join processing on the GPU. We review query processing algorithms on GPUs and CPUs and survey existing techniques on parallel joins.

2.1 Query Processing on GPUs

GPUs have been recently used for general purpose computation such as matrix operation [19] and FFT computation [16]. For additional information on the state-of-the-art GPGPU techniques, we refer the reader to the recent survey by Owens et al. [22]. We now briefly survey the techniques that use GPUs to improve the performance of database operations.

Sun et al. [27] used the rendering and search capabilities of GPUs for spatial selection and join operations. Bandi et al. [4] 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 [11][12] as well as sorting [11], and for data mining operations such as computing frequencies and quantiles for data streams [13]. 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, instead of using graphics APIs, we design parallel join algorithms based on a GPGPU programming model that provides additional flexibility and utilizes hardware resources, such as shared memory among processors, to improve the overall performance of the joins on GPUs.

2.2 In-Memory Query Processing on CPUs

Memory stalls are an important factor for the overall performance of relational query processing [3][7]. Cache-conscious techniques have been the leading approach to improve the memory performance of the CPU joins.

Shatdal et al. [25] proposed the blocked NLJ algorithm by applying cache blocking on the nested-loop join. Rao et al. [23] proposed a cache-optimized B+-tree index, namely the CSS-tree. A CSS-tree has a node size equal to the cache block size and eliminates all pointers in the tree node. Thus, the nodes are fully packed with keys and are laid out contiguously, level by level. LaMarca et al. [18] studied the cache performance for the quick sort and showed that cache optimizations can significantly improve the performance of the quick sort. Boncz et al. [7] proposed the radix hash join with a multi-pass partitioning method in order to optimize the cache performance.

Following these optimization techniques on CPUs, we consider optimizations on the shared memory on GPUs to reduce the stalls on the device memory.

2.3 Parallel Joins

Parallel algorithms greatly improve the performance of the relational join in shared-nothing systems [20][24] or shared-memory systems [8][21].

Liu et al. [20] investigated the pipelined parallelism for multi-join queries. For the algorithm for a single join operation, Lu et al. [21] studied four hash-based join algorithms on multiprocessor computers sharing the main memory. Schneider et al. [24] evaluated one sort-merge and three hash-based join algorithms in a shared-nothing system. Both studies showed that the parallel hybrid hash-join algorithm was superior in the absence of data skews. The parallel hybrid hash join is performed in two phases [24]. First, it distributes the *R* tuples to the processor nodes, where some tuples are inserted into the in-memory hash table and others are spooled to the disk. Next, the *S* tuples are distributed to the processor nodes. Some tuples are used for probing the hash table built in the first phase, and others are also spooled to the disk. In the presence of data skews, techniques such as bucket tuning [24] and partition tuning [17] are used to balance loads among processor nodes. Recently, Cieslewicz et al. [8] implemented a multi-threaded hash join on the Cray MTA-2 architecture.

In this paper, we develop parallel join algorithms running on new-generation GPUs, which are massively multi-threaded SIMD processors with high-bandwidth, high-latency video memory and are newly equipped with on-chip shared-memory for inter-processor communication. Our design and implementation takes into account the GPU architectural characteristics and provides general, yet efficient solutions to GPU-based joins.

3. OVERVIEW

In this section, we describe how we use the GPU in order to efficiently perform primitive operations commonly used in database query processing, particularly in joins. We start by giving a more in-depth overview of the latest GPU architecture, then discuss the challenges of parallel programming on that architecture, and finally present the solutions, which are the building blocks used in our join algorithms.

3.1 GPU Architecture and Programmability

A GPU is a SIMD processing unit with high memory bandwidth primarily intended for use in computer graphics rendering applications. In recent years, the GPU has become extremely flexible and programmable. This programmability is strengthened in every major generation (roughly every 18 months) [22].

Until recently, vertex processing and pixel processing used different sets of customized parallel processing engines. The bulk of the computation in GPGPU applications was often limited to the more numerous and more flexible pixel engines that could easily access the high-bandwidth memories. In contrast, the latest GPUs have a unified architecture with over one hundred processing engines, all of which being capable of processing every programmable step of the rendering pipeline and access all

memories. Such a unified architecture further facilitates the use of the GPU as a general purpose processor.

In addition, many GPU features which are not of paramount importance for rendering applications are now being exposed through separate programming frameworks. Such features are not available through graphics APIs, such as OpenGL and DirectX. In order to best utilize the features of the G80, we use the CUDA (Compute Unified Device Architecture) framework **[2]**, for general purpose computation on GPUs. Unlike DirectX or OpenGL, CUDA provides a programming model for a thread-based programming language similar to C. Thus, database programmers can take advantage of GPUs without requiring graphics knowledge. Two of the newly exposed features - efficient data scattering (i.e., indexed writes to an array structure) and on-chip shared memory for the processing engines, are exploited by our join algorithms.

The hardware model of the GPU is illustrated in Figure 1. At a high level, the G80 GPU consists of many SIMD multi-processors. Each multi-processor consists of 16 processors. At any given clock cycle, each processor of a multiprocessor executes the same instruction from the instruction unit, but operates on different data. The shared memory is a high-bandwidth register file shared by all the processors in a multi-processor. It is similar in nature to the L1 data cache in the CPU. The size of this shared memory is small and the access latency is low. Additionally, the GPU has a large amount of device memory, which has both high bandwidth and high access latency. For example, the G80 GPU has a total shared memory of size 256 KB and device memory bandwidth of 86 GB/s.

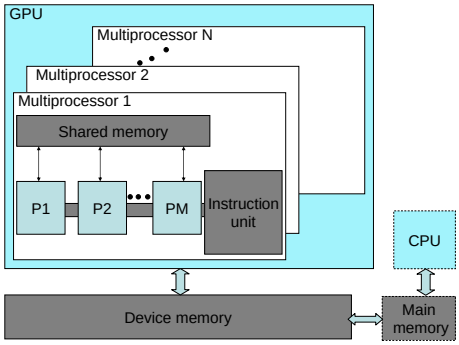


Figure 1. Hardware model of the GPU: A set of SIMD multiprocessors with on-chip shared memory.

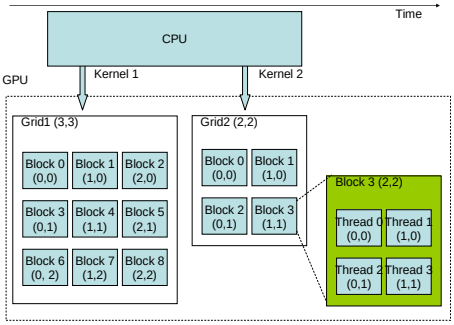


Figure 2. The machine issues two kernels to GPU in succession. Each kernel is executed as a grid of thread blocks.

Table 1. Notations used in this paper

Parameter	Description
Bp	Number of thread blocks per multi-processor
T	Number of threads in a thread block
P	Number of multi-processors in the GPU
M	The average size of shared memory per thread block
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)

3.2 Technical Challenges

Based on the GPU architectural model and the parallel database query processing model, we have identified four technical challenges in parallel programming on GPUs:

- How to efficiently utilize the computation resource and the memory bandwidth of the GPU, and to use the parallel computation to hide memory latency.
- How to handle the read/write conflicts efficiently. Since we do not have hardware support primitives for conflict handling, we need to develop an efficient conflict handling mechanism that is suitable for GPUs.
- How to handle the non-uniform data distribution on GPUs.
- How to best take advantage of hardware dynamic flow control. Within a thread block, if two threads evaluate the branch condition differently, then both sides must be executed. As a result, divergence in the branching of a thread block can lead to reduced performance.

3.3 Primitives

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

1) They utilize the efficient inter-processor communication, achieving 4-10x higher effective memory bandwidth than CPUs.
2) They have low synchronization and branching overhead, thus achieving close to peak performances on GPUs.
3) They are scalable to hundreds of processors.
4) They have deterministic performance irrespective of data skews

To illustrate the performance impact of our primitives, we will include performance comparison figures for each of them in this section. In general, the data sets used in this section are uniformly distributed and are up to the maximum size that allows the processing to be video-memory resident. The details of the experimental setup, experiments on other data sets, and join performance results will be described in Section 5.

3.3.1 Scan

A scan is commonly used in relational query operators. In GPGPU, a scan applies a binary operator to the input relation of size *n* in linear time and generates an output relation of size *n*

[22]. We present the definition of a +Scan which applies the binary operator + to the input relation as follows:

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

We use multiple thread blocks to implement the scan. Each thread block is responsible for a segment of the relation. The scan can take advantage of a hardware feature of the GPU, *the coalesced read* **[2]**. That is, GPU recognizes the consecutive addresses of data accesses from the threads and merges these accesses into a single contiguous memory access. Thus, the coalesced read fully utilizes the hardware bandwidth of GPU and greatly improves the scan efficiency.

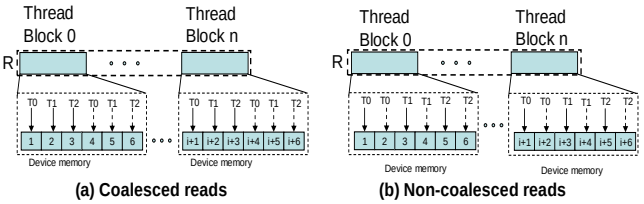


Figure 3. Scans with and without coalesced reads.

Figure 3 illustrates two scan schemes with and without coalesced reads. Suppose a thread block consists of three threads T0, T1 and T2. In Figure 3 (a), 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. For example, T0, T1 and T2 read the location 1, 3 and 5, respectively. This results in a higher number of non-coalesced reads.

Figure 4 demonstrates the scan performance on a sequential scan on CPU and the partitioned scan with and without coalesced reads. We copied all tuples from relation R_{in} to relation R_{out}. Both GPU-based methods are over ten times faster than the CPU-based one. The coalesced scheme further improves the scan on GPU by over five times. With coalesced reads, the GPU achieves a bandwidth of around 70GB/s.

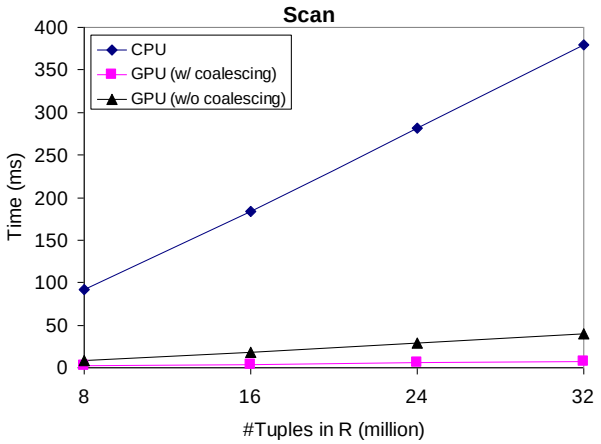


Figure 4. Execution time of scans on GPU and CPU

Computing the prefix sum is an important operation on parallel databases [6]. Given an input relation (or array) R_{in}, each index of the output array $R_{out}[i]$ ($2 \le i \le |R_{in}|$) is obtained from the sum of $R_{in}[1]$,..., and $R_{in}[i\text{-}1]$ ($R_{out}[1]$=0). We used the implementation provided by NVIDIA Corp. [2]. This implementation is based on the parallel prefix sum algorithm proposed by Blelloch [6]. It is work efficient and utilizes the shared memory and coalesced reads from the device memory.

3.3.2 Scatter and Gather

A scatter performs indexed writes to a relation and it is useful for 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, …, n], L [1, …, n].
Output: R_{out} [1, …, n].
Function: $R_{out}[L[i]]=R_{in}[i]$, i=1, …n.

The gather primitive performs indexed reads from a relation. It can be used, for instance, when fetching tuples 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, …, n], L [1, …, n].
Output: R_{out} [1, …, n].
Function: $R_{out}[i]=R_{in}[L[i]]$, i=1, …n.

We apply similar implementation schemes to these two primitives. We first describe the techniques for the scatter in detail, and then present the gather in brief.

A basic implementation of scatter is to scan L and R_{in} once. It outputs all R_{in} tuples to R_{out} during the scan. This basic implementation is simple. However, if L is random, the scatter suffers from the random writes to R_{out}. Due to this randomness, these writes have low spatial locality and are hardly coalesced.

To improve the performance of scatter, we design a multi-pass scheme to improve its spatial locality. The basic idea is that in each pass, we output the R_{in} tuples only when they belong to a certain region of R_{out}. The algorithm first divides R_{out} into a number of chunks (denoted as *nChunk*), and then performs the scatter in *nChunk* passes. In the ith pass ($1 \le i \le nChunk$), it scans L once, and outputs the tuples belonging to the ith chuck of R_{out} (i.e., their write locations are between $(i-1)\cdot\frac{|R_{in}|}{nChunk}$ and $i\cdot\frac{|R_{in}|}{nChunk}$).

Since each chunk is much smaller than $|R_{in}|$, our multi-pass scatter has a better spatial locality than the single-pass scatter.

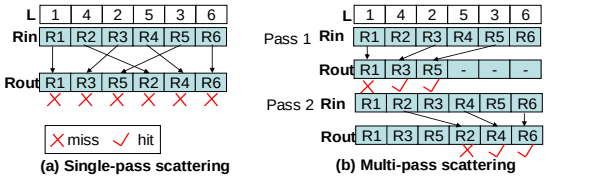


Figure 5. Single-pass vs. multi-pass scattering: the multi-pass scattering has a better locality than the single-pass scattering.

Let us illustrate the effectiveness of our multi-pass scheme using an example, as shown in Figure 5. For simplicity, we assume that the GPU can hold only one memory block and a block can hold

three tuples. Each write in the single-pass scheme results in a miss. The single-pass scheme has a low spatial locality. In comparison, the multi-pass scheme divides S into two chunks, and writes a chunk in each pass. It achieves an average miss rate of 1/3.

We develop a cost-based model for determining the suitable value for *nChunk*. Let the bandwidths of the scan and random accesses be B and B' (bytes/sec), respectively. The cost for our multi-pass scatter is estimated as the total cost of sequential scans and random writes, i.e., $nChunk \cdot \frac{\|R_{in}\|}{B} + \frac{\|R_{in}\|}{nChunk \cdot B'}$. When $nChunk = \sqrt{\frac{B}{B'}}$, the multi-pass scatter has the minimum cost. On G80, the suitable *nChunk* value is 8 based on our model.

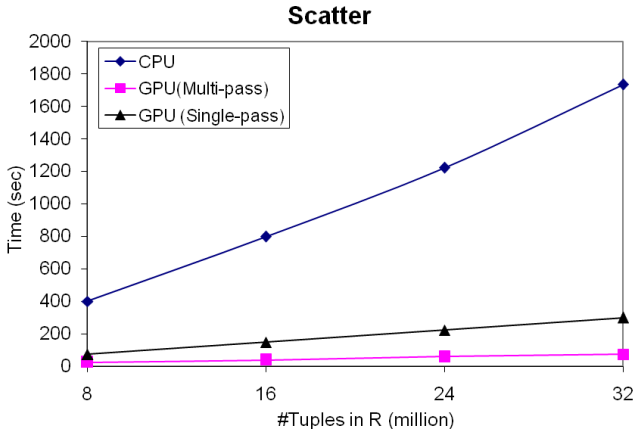


Figure 6. Execution time of scatter on GPU and CPU

Figure 6 demonstrates the performance of scattering on GPU and CPU. Note that the write locations of R are randomly generated, which simulates the worst case spatial locality for scattering. Our multi-pass is over three times faster than the single-pass scatter. It achieves a bandwidth of 6.6 GB/s, which is more than 15 times faster than the CPU-based scatter.

As with scatter, random accesses in the gather primitives significantly degrade performance. So, when L is random, we use a multi-pass scheme that is similar to the one in the scatter. The difference is that we read a chunk of R_{in} in a pass for gather instead of writing a chunk of R_{out} in a pass for scatter. Figure 7 shows the performance comparison of gather on GPU and CPU. The GPU-based multi-pass gather is 16 times faster than its CPU-based counterpart, and is about three times faster than the GPU-based single-pass gather.

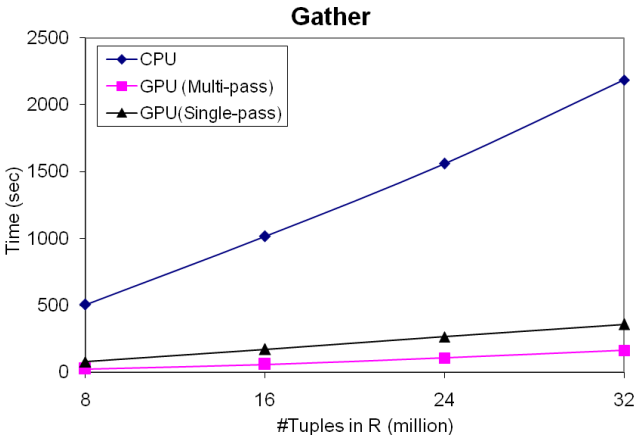


Figure 7. Execution time of gather on GPU and CPU

3.3.3 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. The definition is as follows:

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

A major issue is the read/write conflicts. They occur when multiple threads try to insert tuples into a partition concurrently. Unfortunately, there is no exposed hardware support for handling such conflicts. In addition to handling such conflicts, our split algorithm takes advantage of the two features of GPU: the fast scan and the efficient scatter.

Our split algorithm computes the write location for each tuple (stored in the array L) and scatters R_{in} to R_{out} according to the array L. We use histograms to compute the location array L. Our algorithm is partially inspired by the parallel radix sort proposed by Zagha [28], which uses histograms to perform the radix sort. The major difference is that we fully consider the hardware features of GPU to design our histogram scheme.

Each thread block is responsible for a portion of R_{in}. The algorithm maintains three levels of histograms: a thread-level histogram for each thread (*tHist*), a thread block-level histogram (*bHist*) for each thread block, and a global histogram (*gHist*). They represent the number of tuples within a portion of the relation belonging to each partition.

Additionally, the algorithm maintains three kinds of offset information: thread-level ($tHist\_sum[1,\ldots, T][1,\ldots, F]$), thread block-level ($bHist\_sum[1,\ldots, Bp \cdot P][1,\ldots, F]$) and global offsets ($gHist\_sum[1,\ldots, F]$). $gHist\_sum[i]$ stores the start location to write partition i. $bHist\_sum[b][i]$ is the offset to $gHist\_sum[i]$ for writing the tuples belonging to partition i processed by thread block b. $tHist\_sum[t][i]$ represents the offset to $bHist\_sum[b][i]$ (thread t belongs to thread block b) for writing the portion of tuples belonging to partition i processed by thread t. These offsets

are computed as prefix sums on the histograms of their corresponding levels.

Given these offsets, we can determine the write location for a tuple. Suppose a tuple is the *x*th tuple belonging to partition *i* processed by thread *t* in thread block *b*. The write location for this tuple is (*gHist_sum*[*i*]+ *bHist_sum*[*b*][*i*]+*tHist_sum*[*t*][*i*]+*x*+*1)*. We denote (*gHist_sum*[*i*]+ *bHist_sum*[*b*][*i*]) to be *bOffset*[*b*][*i*].

The algorithm works in five steps.

Step 1) Each thread constructs its *tHist* histogram.

Step 2) Each thread block constructs its *bHist* histogram and the thread-level offsets, *tHist_sum*, by computing the prefix sum against all the thread-level histograms of the thread block. Then, each thread updates a portion of the *L* array with the index of the tuple belonging to a certain partition.

Step 3) The algorithm constructs the histogram *gHist* and the block-level offsets, *bHist_sum*, by computing the prefix sum against all the thread block-level histograms. The global offsets *gHist_sum* are computed as a prefix sum on *gHist*.

Step 4) Each thread block computes *bOffset* based on *gHist_sum* and *bHist_sum*. Then, each thread updates a portion of *L* based on *bOffset.*

Step 5) *R* is scattered to *S* according to *L*.

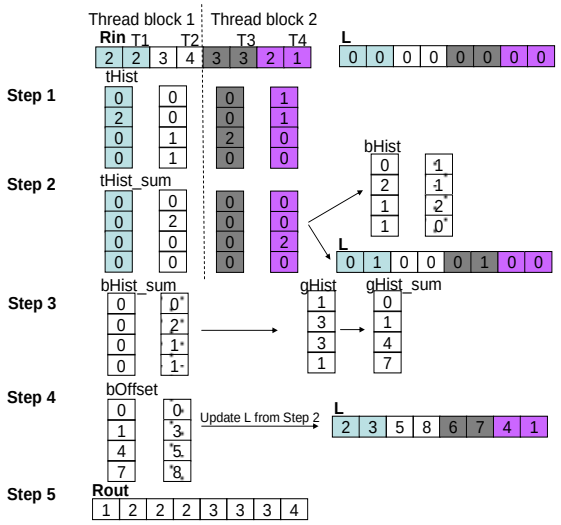


Figure 8. An example of the split primitive.

Figure **8** shows an example of our split operation, where we divide R_{in} into four partitions. In this example, there are two thread blocks, one containing T1 and T2 and the other one containing T3 and T4. The portions of R_{in} processed by different threads are in different colors. In the first step, there are four thread-level histograms. Step 2 creates two thread block-level histograms and updates the *L* array. Step 3 creates a global histogram. Step 4 creates two *bOffset* arrays for updating *L*. Since the write location is deterministic, tuples can be output in parallel.

Since the thread-level histogram is accessed randomly in the first step, we store these histograms into the shared memory. Suppose the number of partitions is *f*. In our implementation, each thread-level histogram is encoded in a char array. Each thread is responsible for 255 tuples at most. We determine *F* to be the maximum *f* such that $T \cdot f \le M$. With this optimization, the split operation performs three scans on R_{in}, two scans on *L* and one scan on R_{out}.

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 determine the number of partitions generated in each level of recursion.

Figure 9 shows the performance of the split primitive on GPU and CPU. We fixed |R| to be 16 million and varied the number of partitions, *F*. "GPU" and "GPU (w/o opt)" denotes the split primitive storing the histograms in device memory and in the shared memory, respectively. The optimization on the shared memory greatly improves the performance of the split on GPU. Due to the increase in the total histogram size, the GPU-based split is slower than the CPU-based one when the number of partitions increases. In contrast, the optimized GPU-based split is around twice faster than the CPU-based one.

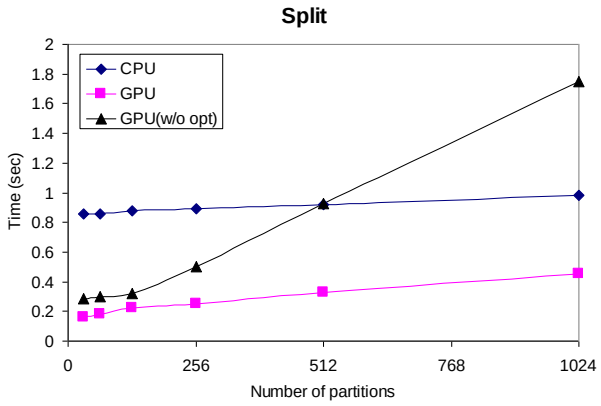


Figure 9. Execution time of split on GPU and CPU

3.3.4 Sort

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

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

We use an extension of the bitonic sorting network [5], because independent swaps between the elements in this sorting algorithm map well to the massively threaded architecture of GPU. For self-contentedness, we briefly describe the basic algorithm before introducing our optimizations.

The bitonic sort merges *bitonic sequences* in multiple stages. A bitonic sequence is a monotonic ascending or descending sequences. Given a relation R_{in}, the bitonic sorting algorithm has $\log_2|R_{in}|$ stages. Stage *x* has *x* steps ($1 \le x \le \log_2 |R_{in}|$). 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. An example of bitonic sort on eight integers is shown in Figure 10. After three stages, these eight integers are

sorted. Tuples at the head and the tail of an arrow are compared, and the larger one is moved to the head of the arrow.

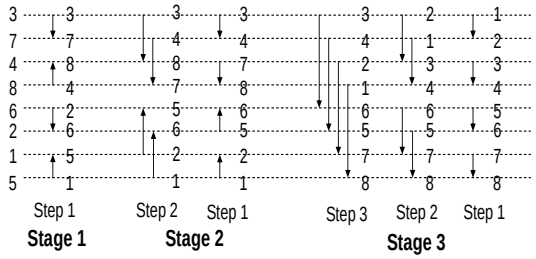


Figure 10. Bitonic sort on eight integers.

A straightforward implementation of bitonic sort is to store the input relation in the device memory and perform the bitonic sort on the device memory directly. Its data access pattern is shown on the top of Figure **11**. The basic implementation fully utilizes the hardware bandwidth. However, it performs repetitive fetches from the device memory, and does not utilize the faster shared memory.

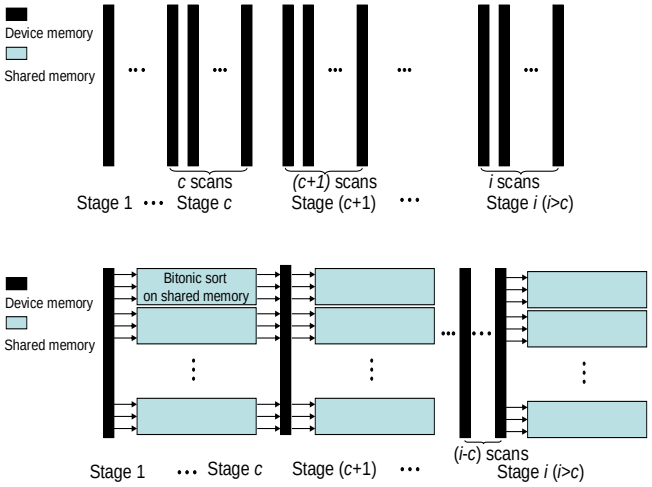


Figure **11**. **Data accesses in bitonic** sort on GPU ($c=log_2\frac{M}{r}$ **):**

(top) without optimizations, (bottom) with optimizations.

Analyzing the access pattern in the bitonic sort, we propose two optimization techniques on the shared memory in order to reduce the number of scans in the basic implementation.

- Optimization 1: The first technique optimizes the first c stages ($c = \log_2 \frac{M}{r}$). The first c stages of the bitonic sort are independently performed on individual chunks of size M. For example, Stages 1 and 2 in Figure **10** is performed on a chunk of four elements. Thus, we use shared memory to store this chunk of data and process the Stage 1 to Stage c on the shared memory.
- Optimization 2: The second technique applies to Steps 1, 2, …, and c in Stage i ($i > c$). The first c steps of Stage i sort a bitonic sequence of size M. We store this bitonic sequence into the shared memory at the (i-c)th step so that Steps 1, 2, …, and c process the data in the shared memory. Steps (c+1), (c+2), …, and i do not use the shared memory and directly handle the data from the device memory.

With these two optimizations, the number of scans is greatly reduced. The first optimization has only one scan. It saves $\#OPT_1=2+\ldots+c=\frac{1}{2}c\cdot(c+1)-1$ scans compared with the basic implementation. The second optimization saves *(c-1)* scans on Stage i ($i > c$), which saves $OPT_2=(c-1)(log_2|R_{in}|-c)$ scans in total.

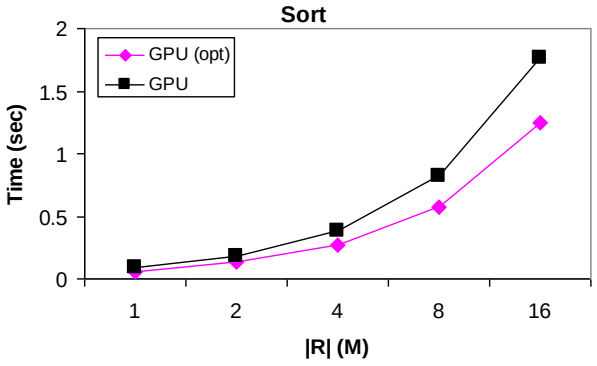


Figure 12. Execution time of bitonic sort on GPU

Since the previous work [11] has shown that the GPU-based bitonic sort dominates CPU-based sorting algorithms, we only show the GPU-based bitonic sort with or without the shared-memory optimization in Figure 12. The algorithm with optimizations using the shared memory is over 40% faster than the one without.

4. JOIN ALGORITHMS

We now briefly describe our join algorithms in detail, including the non-indexed and indexed nested-loop join (NINLJ and INLJ respectively), the sort-merge join (SMJ) and the hash join (HJ). Since these algorithms are well known in the literature, we focus on the differences in our GPU-based implementations, especially their usage of our primitives. Specially, NINLJ uses the scan primitive on both relations; INLJ uses a scan primitive on the outer relation and a gather for probing the tree index; SMJ uses the sort on both relations and next scan the sorted relation for merging; HJ uses the split primitive followed by the scan on both relations.

4.1 Non-indexed NLJs (NINLJ)

The nested-loop join can be naturally mapped to the programming model of CUDA, as shown in Figure 13. The dots represent tuples generated by the join, some of which may be eliminated by the join predicate.

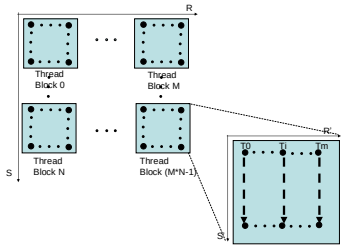


Figure 13. The non-indexed NLJ algorithm on GPU

Our algorithm is blocked nested-loops. Each thread block computes the join on a portion of *R* and *S*, denoted as *R'* and *S'*, respectively. Within a thread block, each thread processes the join on one tuple from *R'* and all tuples from *S'*. Thus, the number of threads in each thread block is equal to the number of tuples in *R'* ($|R'|=T$).

We store *S'* into the shared memory to avoid reading *S'* repeatedly from the device memory. Due to the limited size of the shared memory on each multi-processor, the size of *S'* is set to be the average size of shared memory for each thread block, i.e., $||S'|| = M$. Since each thread block requires to scan *R'* and *S'* only once from the device memory, the total volume of data transfer between the GPU and the device memory is $\frac{|R|\cdot|S|}{|R'|\cdot|S'|}(||R'||+||S'||)=\frac{|R|\cdot|S|}{T\cdot M}(T\cdot r\cdot s+M\cdot s)$.

To avoid conflict where multiple threads write results to the shared output region in the device memory, we perform the join result output in three steps. In the first step, each thread counts the number of join results for the small 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. In the second step, 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. In the third step, each thread outputs the join results to the device memory according to its start write location. Since this output scheme is similar in other joins, we will omit it in the remainder of this section.

4.2 Indexed NLJs (INLJ)

We implement the indexed join algorithm through adapting the cache-optimized search tree, CSS-Tree **[23]**, 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 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 scanning 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. 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 is used to locate the index of the node to go.

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

While searching for a single key in such a tree offers little opportunity for parallel processing, multiple searches, however, fits extremely well into the parallel programming model. 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. This parallel search is implemented as a gather: at each level of the tree, we maintain an array *L*, where $L[i]$ ($1 \le i \le |R|$) is the tree node index that the *i*th probe will go to. All L[i] are initialized as the index of the root node. At each level, we use a gather primitive to fetch the node that the *i*th probe will go to according to $L[i]$. A binary search is used to locate the index of the node to go.

4.3 Sort-Merge Joins (SMJ)

Similar to the traditional sort-merge joins, we first sort the two relations and next perform a merge step on these two sorted relations.

After both relations are sorted, we merge them in parallel. We identify the *Q*-quantile on the smaller relation *S*. We set *Q* to be $\frac{||S||}{M}$. We use *Q* thread blocks to perform the merge step so that *S* is uniformly distributed to the *Q* thread blocks. Thus, we can identify the partition pairs of *R* and *S*. We load an *S* partition into the shared memory and perform the merge join on this *S* partition and the corresponding *R* partition. Within a thread block, each thread is responsible for a range of tuples from the *R* partition, and scans the tuples from the *S* partition for matching.

When *R* and *S* are uniform, the *Q*-quantile divides *R* and *S* into *Q* partitions of uniform sizes. Otherwise, the imbalanced partition size hurts the performance due to the SIMD nature of GPUs. Therefore, we handle the merge joins according to the partition size in the merge step. We first handle the "small" joins with partitions whose sizes are smaller than *M*. Note, these small joins are evaluated in parallel. The average execution time for these small joins is that of computing a nested-loop join on an *R* partition of $\frac{M}{r}$ tuples and an *S* partition of $\frac{M}{s}$ tuples. Next, we evaluate the "large" joins. Each of these large joins has at least one input partition of a size larger than *M*. If a partition is larger than *M*, we divide it into multiple chunks each of size *M*. We then apply NINLJ to iterate over these chunks.

4.4 Hash joins (HJ)

We develop a parallel version of the radix hash join **[7]**. Our algorithm has two phases.

Phase 1) Partitioning. We split *R* and *S* into multiple partitions so that one *S* partition can fit into the shared 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. A NINLJ is used to evaluate the join of the partition pair. We use the same skew handling scheme as the one in SMJ.

5. EXPERIMENTS

In this section, we evaluate the performance of our proposed GPU join algorithms in comparison with the joins on CPUs.

5.1 Experimental Setup

We have implemented and tested our algorithms on a PC with a G80 GPU and an Intel P4 Dual-Core processor running Windows XP. 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 bandwidth of 2 GB/s.

Table 2. Hardware configuration

	GPU	CPU
Processors	1350MHz × 8 × 16	3.2GHz (Dual-core)
Data cache (shared memory)	16KB × 16	L1: 16KB, L2: 512KB
Cache latency (cycle)	2	L1: 2 , L2: 10
DRAM (MB)	768	1024
DRAM latency (cycle)	200	300
Bus width (bit)	384	64
Memory clock (GHz)	1.8	0.8

We compute the theoretical bandwidth to be the bus width multiplying the memory clock rate. Thus, GPU and CPU have a theoretical bandwidth of 86.4 GB/s and 6.4 GB/s, respectively. Based on our measurements, for scans, the G80 achieves a memory bandwidth of around 69.2 GB/s whereas Intel P4 has 1.3 GB/s. A GPU scan on a relation of size 512MB completes in less than 10ms. For scatter, the G80 has a memory bandwidth of around 6.6 GB/s whereas the Intel P4 has 0.4 GB/s. The GPU is over an order of magnitude faster than the CPU for these two operations. Compared with the CPU, the GPU can boost the performance of data-intensive applications such as relational query processing.

We used our homegrown 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>;
```

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 $R.key \le S.key \le R.key + \delta$. In order to evaluate our algorithms with different join selectivities, we changed the percentage of tuples having matches in one relation for the equi-join and varied the δ value in the non-equijoin predicate.

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; for other three joins, we fixed |*R*| to be 16 million. Second, we examined the performance impact of join selectivity. Third, we investigated our algorithms on the non-uniform data sets. In the other two sets of experiments, we fixed both |*R*| and |*S*| to be one million for NINLJ. For 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 comparable to the previous studies on in-memory join algorithms [7].

Finally, we varied the δ value in the non-equijoin predicate and examined the performance of non-equijoins.

For each GPU join, we transfer data from the main memory to the device memory, process the join on the GPU, and copy results from the device memory to the main memory.

Implementation details on CPU. For comparison, we have implemented highly optimized CPU-based join algorithms including the blocked NLJ [25], the indexed NLJ with the CSS-tree index [23], the sort-merge join with the optimized quick sort [18] and the radix hash join [7]. The suitable values for the cache parameters in the algorithms, such as the tree fanout of the CSS-tree, are tuned according to the cache configuration of the CPU. We compiled our algorithms using Intel compiler 9.1 with full compiler optimizations [1]. These optimizations included vectorization, where the compiler generates SIMD execution for sequential data scans, and multi-threading.

Implementation details on GPU. To implement an efficient algorithm on the GPU, we need to determine the following parameters with respect to the target operation: the number of threads for each thread block (*T*) and the number of thread blocks per multiprocessor (*Bp*).

Issuing more threads to the GPU can potentially improve the overall performance by hiding memory latency. However, due to the limited shared memory space on the multiprocessors, *Bp* cannot be arbitrarily large. Through experiments, we find that *Bp*=16 is a good tradeoff value, where the memory latency is sufficiently masked, and each block receives adequate shared memory space. The *T* value was chosen with respect to the target algorithm. In our experiments, we set *T* to multiple warp size for high utilization of the computation resources, where the warp size is the number of threads that can be scheduled each time on a multiprocessor (32 on G80).

Given *Bp*=16, the average shared memory size is 1KB (or 128 tuples). NINLJ sets *T* to be 128 such that each thread is responsible for one tuple. We also use this *T* value for the merge step in SMJ and the matching phase in HJ. 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. Thus, INLJ stores only the root node into the shared memory, because the total size of nodes in the upper two levels exceeds 1KB. The bitonic sort sets the chunk size for the optimization on the shared memory to be 1KB (c=8).

5.2 Scatter

We consider three parameters in the evaluation of our scattering algorithm: the relation size (||R||), the distribution of writing locations (L), and the number of chunks on S (nChunk). To simulate different distributions of L, we first divide R into #p partitions using a function R[i] mod #p. We varied the #p value to obtain different distributions for L.

We first fixed the #p value to be 64 and evaluated the performance impact of nChunk. The result is shown in Figure 14 when |R| is 8M, 16M and 32M. The performance curve is concave. As the number of chunks in the output relation is smaller than a threshold value of *nChunk* (eight in this figure), the scatter performance improves because the spatial locality is improved. As the number of chunks is larger than the threshold value, the overhead of extra scans becomes significant. Additionally, the threshold value of *nChunk* is not affected by the relation size. This indicates the accuracy of our cost model on the suitable *nChunk* value.

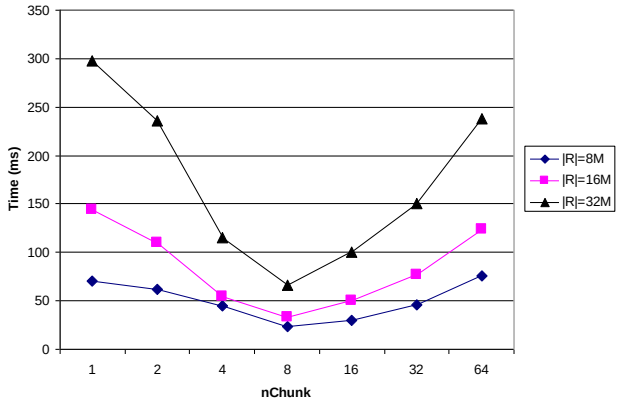


Figure 14. The performance of scattering on GPU: #p=64.

We next varied the #p value to evaluate its impact. The result is shown in Figure 15. When $\#p \le 8$, the suitable value of nChunk is one; when $\#p$=16, the suitable value is two; when $\#p = 32$, the suitable value is four; when $256 \le \#p \le 1024$, the suitable value is 16; for other cases, the suitable value is 8. Our model does not predict all these values correctly, because our cost model assumes the uniform distribution on the write location in the scatter. Nevertheless, the suitable value for nChunk is no larger than 16 regardless of the #p value. Figure 16 shows the execution time of scattering on GPU and CPU. The GPU has a scattering bandwidth of 6.6 GB/s, which is more than 15 times faster than the CPU.

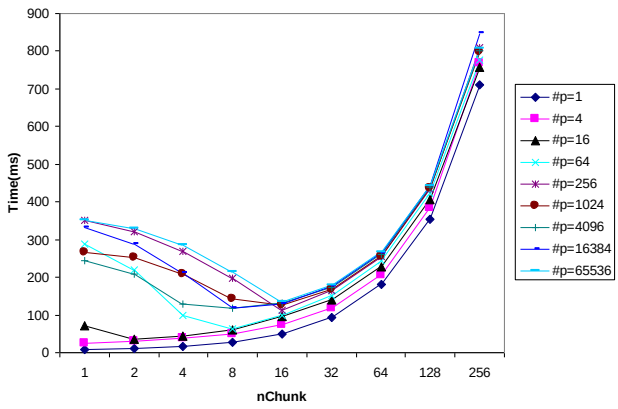


Figure 15. The performance of scattering on GPU: |R|=32M.

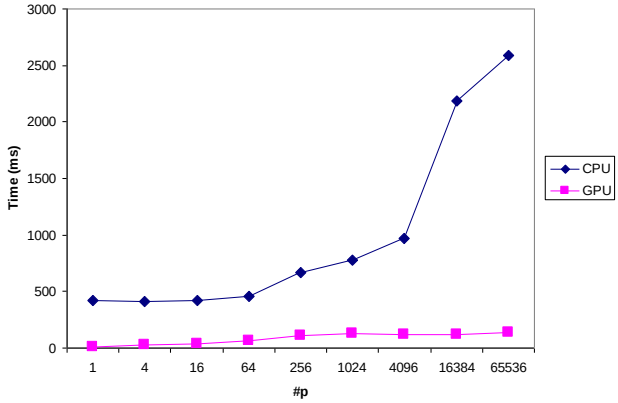


Figure 16. Execution time of scattering on GPU and CPU.

5.3 Results on joins

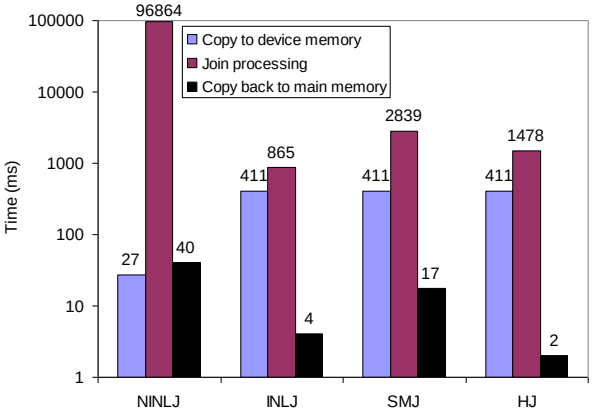


Figure 17. Join processing time and data transfer time between the main memory and the device memory

Prior to our detailed study on our join algorithms, we investigated the join processing time and the data transfer time between the main memory and the device memory. The results for our four

joins under the default setting are shown in Figure 17. For all join algorithms, the join processing time is dominant and the cost of data transfers between the main memory and the device memory is negligible. It is the major component for performing joins on the GPU. In the remainder of this study, we focus on the join processing time.

5.3.1 Non-indexed NLJs

Figure 18 shows the execution time of non-indexed NLJs on the equijoin query and the non-equijoin query. We also compare the GPU algorithms with and without caching the *S* tuples using the shared memory. This experiment indicates that both variants of the non-indexed NLJ algorithm on GPU are faster than the CPU-based algorithm. Additionally, the shared memory greatly improves the performance of our GPU non-index NLJ. The GPU algorithm with caching is around 19 times faster than the CPU algorithm. We changed the join selectivity and the data distribution for the equijoin, and find that both GPU- and CPU-based algorithms are stable.

We also changed the δ value of the non-equijoin predicate in order to vary the number of matches per R tuple. The GPU-based algorithm is stable due to the high scatter bandwidth on GPU. The GPU algorithm is around 15 times faster than the CPU algorithm. The speedup on the non-equijoin is less than the equijoin. One possible reason is that the branches due to the non-equijoin predicate hurt the performance of the SIMD processors.

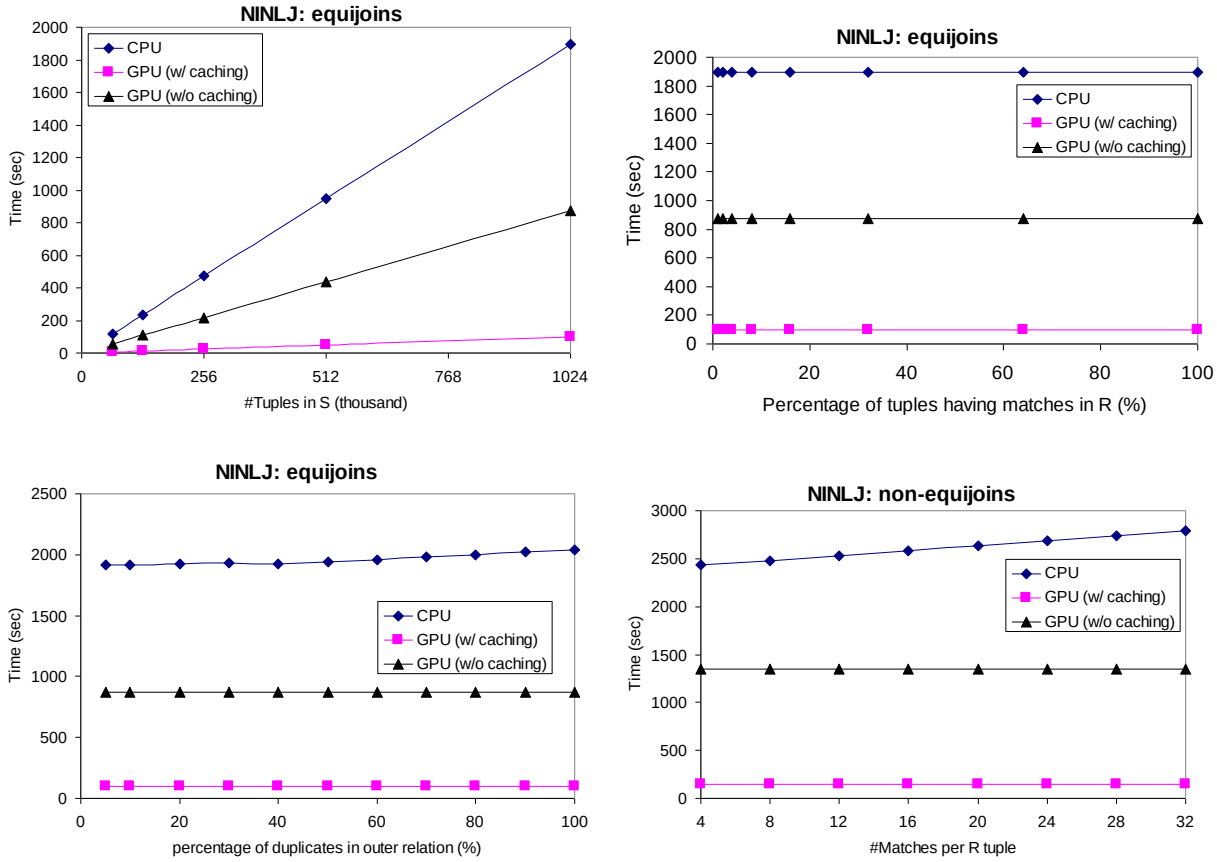


Figure 18. NINLJ on GPU and CPU

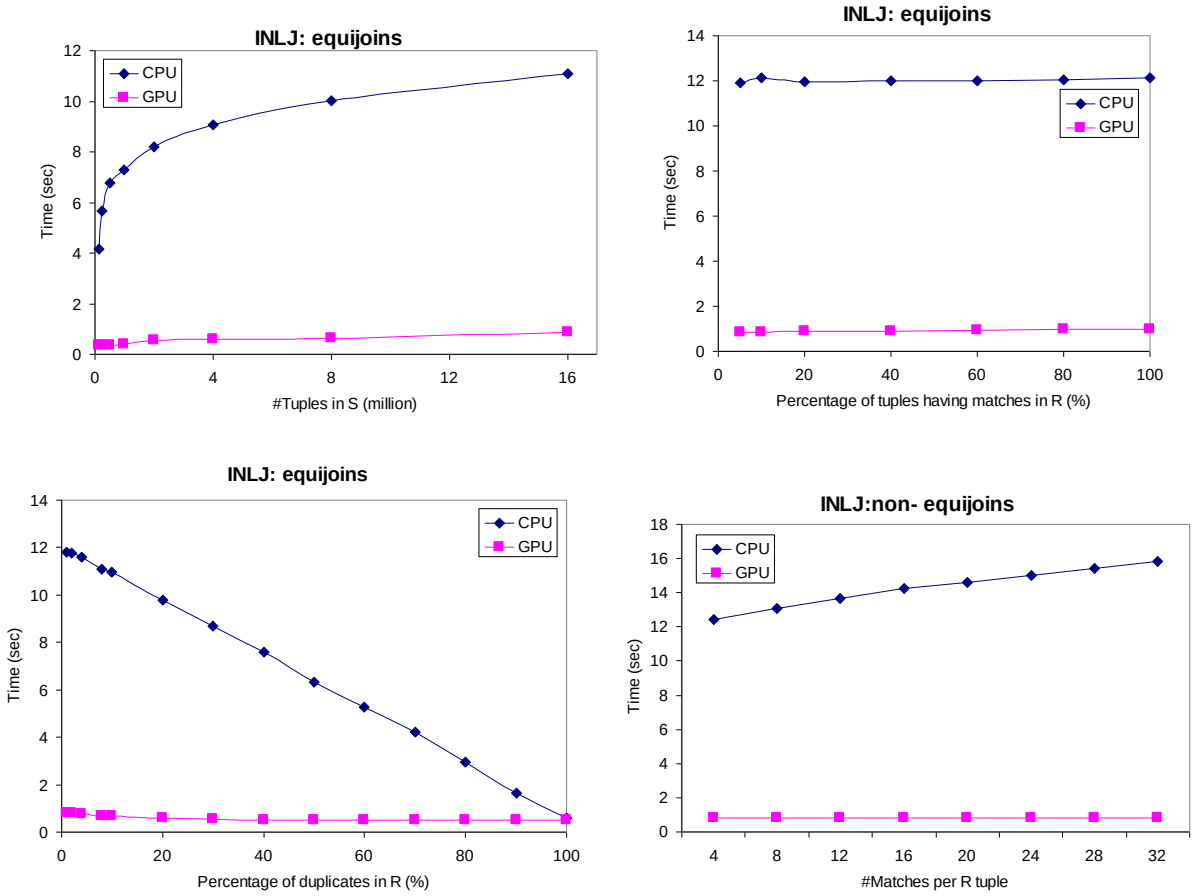
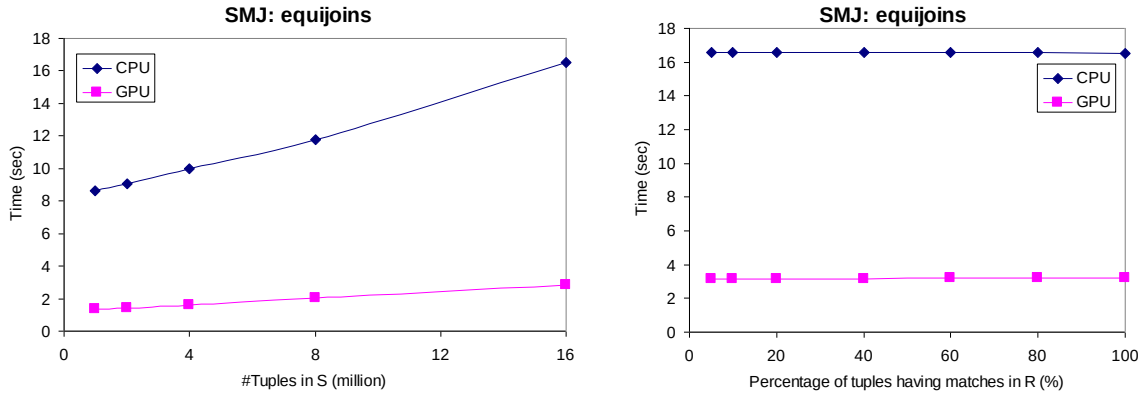


Figure 19. INLJ on GPU and CPU



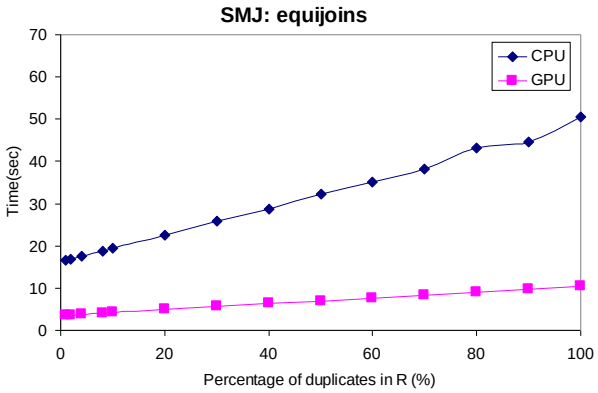


Figure 20. SMJ on GPU and CPU

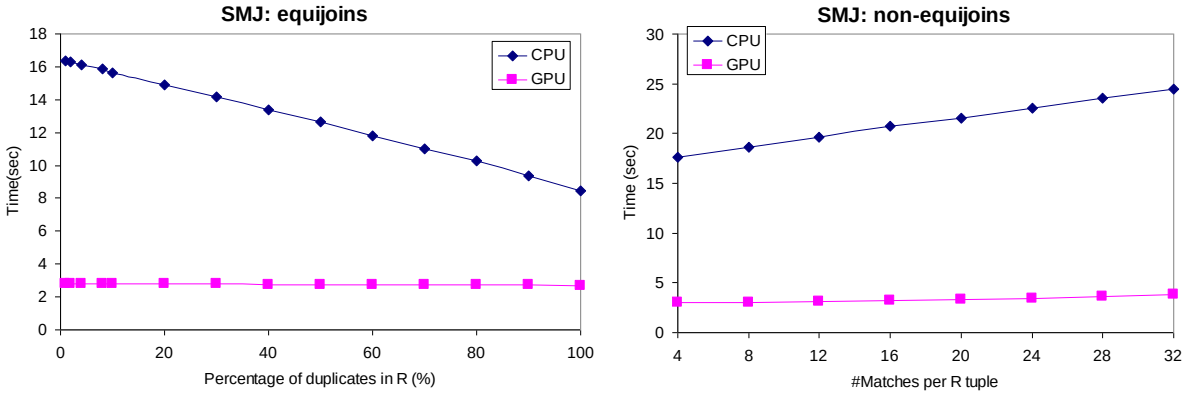


Figure 21. SMJ on GPU and CPU: (left) S is uniform; (right) S is fixed to have 0.01% duplicates

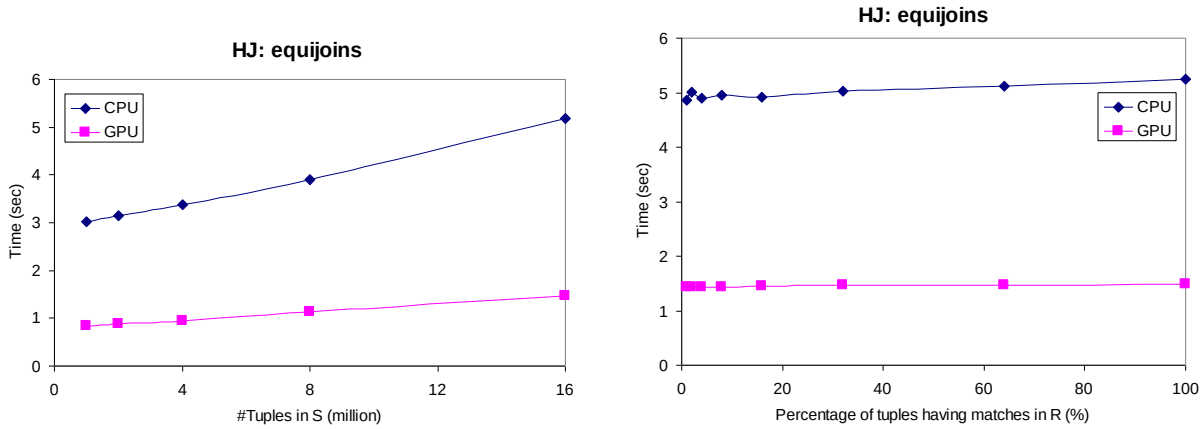


Figure 22. HJ on GPU and CPU

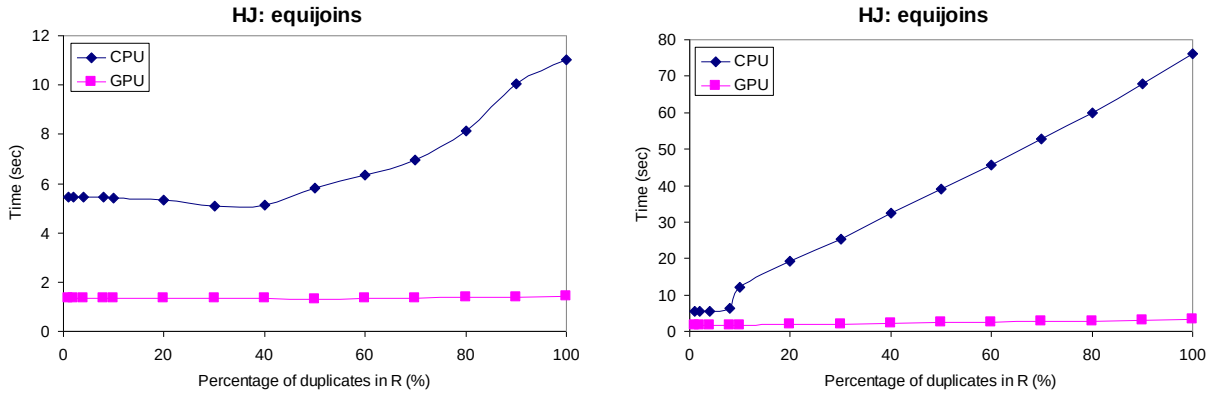


Figure 23. HJ on GPU and CPU: (left) S is uniform; (right) S is fixed to have 0.01% duplicates

5.3.2 Indexed NLJs

Figure 19 shows the execution time of INLJ on GPU and CPU. We observed that INLJ on the GPU is around 16 times faster than its CPU counterpart. The performance improvement is smaller than that of NINLJ. One possible reason for the lower speedup is the branch penalty of the binary search within a tree node. For non-equijoins, the GPU-based INLJ algorithm is around 10 times faster than its CPU-based counterpart. Similar to the NINLJ, INLJ has a smaller performance speedup on the non-equijoin than that on the equijoin. Therefore, figures for these results are omitted.

We investigated the performance impact of data skews in INLJ. When the data becomes more skewed, the spatial and temporal locality of the tree increases. This locality increase explains why the execution time decreases as the number of duplicates in the outer relation increases. The decrease in the execution time on the CPU is larger than that on the GPU. When the data is extremely skewed, the working set becomes very small. It fits into the L2 cache, and even into the L1 cache. Note that the data set is automatically cached in the CPU caches. However, it is not in the GPU. The INIL suffers from the thrashing at the bottom levels of the index tree on the GPU. It is therefore desirable to identify the hot data in the INLJ and to store the hot data into the shared memory for future accesses.

5.3.3 Sort-merge joins

We evaluated SMJ with different data sizes and join selectivities on the uniform data sets for equijoins. We also evaluated SMJ on non-equijoins with the δ value varied. The results are shown in Figure 20. Regardless of the data sizes and join selectivities, the GPU-based SMJ is over six times faster than its CPU-based counterpart. The GPU-based SMJ non-equijoin is around five times faster than its GPU-based counterpart. Similar to the NINLJ and INLJ, the GPU-based SMJ on non-equijoins has a smaller performance speedup than on equijoins due to the branches in the predicate.

We further study the performance of SMJ on equijoin queries with different data distributions. We show our results in two cases in Figure 21: (1) *R* is skewed and *S* is uniform, which is equivalent to the case where *R* is skewed and *S* is uniform; (2) both *R* and *S* are skewed. In the first case, the execution time of GPU-based SMJ is stable. This is mainly because the performance of the GPU-based bitonic sort is stable regardless of data distribution, which accounts for around 90% of the total execution time. In contrast, the execution time of the CPU quick sort decreases, as the data values become skewed. This is because the number of tuple movement decreases in the CPU quick sort. In the second case, the cost of nested-loops in the merge step becomes dominant. We also varied the skewness of *S*. As *S* becomes more skewed, the performance gap between the GPU- and CPU-based algorithms becomes larger. This gap indicates the effectiveness of our skew handling.

5.3.4 Hash joins

We evaluated the hash join with different data sizes and join selectivities on the uniform data sets. The results are shown in Figure 22. The GPU-based HJ is around three times faster than its CPU-based counterpart.

We show the performance of the hash join with non-uniform data sets in Figure 23. We varied the data distributions in two cases similar to those in evaluating the SMJ: (1) *R* is skewed and *S* is uniform; (2) both *R* and *S* are skewed. In the first case, the execution time of HJ on CPU dramatically increases as *R* becomes skewed. In contrast, the GPU-based HJ has a stable performance. In the second case, when *R* is skewed, the execution time of the CPU-based HJ dramatically increases due to the dominant cost of the nested-loops in the matching phase. The execution time of our GPU-based algorithm increases relatively slowly due to the effective skew handling scheme.

Finally, we investigate the time breakdown of our join algorithms. We summarize the results for the joins on uniform data sets under our default experimental setting. The time breakdown of the join processing time on the GPU is shown in Figure 24. Time breakdown of the four GPU joins with and without cache optimizations. We divide the execution time into two components, the memory stalls and the busy time. We estimate the memory stalls to be the total data transfer time between the GPU and the device memory. Note that we distinguish the sequential and the random accesses in the data transfer. The busy time is obtained by subtracting the memory stalls from the total execution time.

Without optimizations on the shared memory, GPU joins suffer from the memory stalls due to the high memory latency. This indicates that spatial and temporal locality of data accesses is important even with the high memory bandwidth of GPU. In contrast, our optimization greatly reduces the memory stalls in

each join algorithm. Our cache optimization techniques reduce the memory stalls by 99%, 20%, 47% and 62% on NINLJ, INLJ, SMJ and HJ, respectively. Among the four join algorithms, the performance improvement of INLJ is smaller than the other three joins. This is because random accesses on the tree structure have low cache locality, which is severe due to the tiny size of the shared memory on the GPU.

5.4 Summary

In summary, our GPU-based parallel primitives and the join algorithms based on them are 2-20x faster than their optimized CPU-based counterparts. We evaluated our join algorithm across different data sizes join queries, join selectivities and data distributions. The performance speedup for the non-indexed NLJ, the indexed NLJ, the sort-merge join and the hash join is up to 19x, 16x, 6x and 3x, respectively.

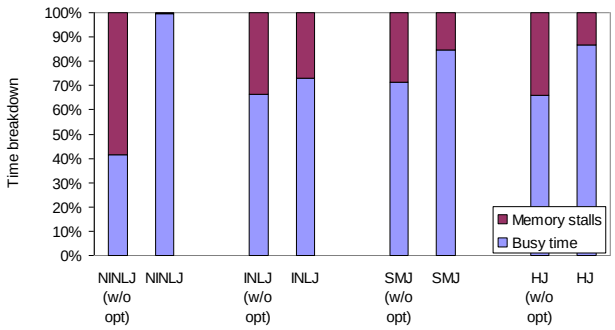


Figure 24. Time breakdown of the four GPU joins with and without cache optimizations

6. DISCUSSION

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 three major opportunities:

First, GPUs have a highly parallel hardware architecture that fits extremely well with data-parallel query processing. GPUs consist of hundreds of in-order processors with the ability to execute several thousands of threads, thus hiding the memory latency more efficiently than traditional von Neumann architectures. Moreover, the high memory bandwidth in GPUs can significantly accelerate the performance of many database operations.

Second, the GPU programmability for general-purpose computing has been improving greatly. The CUDA from NVIDIA Corporation and the CTM programming API from AMD Corporation demonstrate the GPU hardware vendors' great amount of effort in extending the functionality of GPUs for the general computing market in addition to the traditional gaming market. In the CUDA programming API, the programmer can program the GPUs without any knowledge of graphics rendering APIs. Instead, it exposes a general-purpose, massively multi-threaded parallel computing architecture and provides a programming environment similar to multi-threaded C/C++. This programmability allows us to develop efficient query processing primitives that utilize the GPU hardware features easily.

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 have proposed a set of data-parallel primitives that share the philosophy of high-level data-centric abstraction of GPGPU. Moreover, we adapt CPU-based optimization techniques to the GPU hardware features, especially the memory hierarchy, to improve the overall performance of relational query processing on GPU. For instance, to improve the temporal locality in split and sort, we store frequently accessed data in the shared memory to be shared within a block of threads; to improve the spatial locality in scan and scatter, where the temporal locality is low, we arrange the access patterns to achieve sequential access among the threads. These techniques effectively reduce the latency through the shared memory and increase the effective memory bandwidth.

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. In our implementation, we rewrite the branches such that most of the time, all processors are executing the same branch concurrently. 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. Nevertheless, we wonder whether it would be feasible for GPUs to add some simple but effective branch prediction mechanisms to the hardware or runtime for representative GPGPU applications, as CPUs already do for common workloads. Alternatively, adopting the MIMD (Multiple Instruction Multiple Data) architecture may be a reasonable choice to avoid such issues.

Another example is that the synchronization mechanism for handling read/write conflicts, which happen constantly in query processing, is limited in the GPU. It would be more flexible if GPUs provide hardware support for conflict handling as an optional choice for applications to take or not, as does in Cray MTA-2. However, the downside of having this flexibility is the performance overhead of maintaining cache coherence among hundreds of processors, which also results in more complex chip design.

Second, with the exposure of the massively multi-threaded hardware architecture and the introduction of a general and powerful multi-threaded programming model through CUDA, 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, if the hardware vendors provide such primitives, which would presumably better

utilize hardware features and run faster than application-level software solutions. This will also alleviate some of the issues of synchronization mentioned above, since they usually occur in these low-level components.

Finally, even though new generation GPU architectures, such as 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. For instance, some of the GPU hardware details, e.g., the texture memory and cache design, are unclear, which would have a great performance impact when exposed and utilized effectively. Currently, we mainly rely on empirical experiments to estimate the hardware parameters and to identify the suitable settings for our algorithms. Additionally, as CUDA is in a preliminary stage, its runtime environment and performance seems to bear a certain degree of uncertainty. Nevertheless, based on our experience and observations in the CUDA's developers' network, we expect this framework to grow and mature to better enable GPGPU.

7. CONCLUSION

Graphics processors have become an attractive alternative for general-purpose high performance computing on commodity hardware. The continuing advances in the GPU hardware and the recent improvements on the GPU 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 correspondingly. We find that our baseline GPU joins are 2-5 times faster than their optimized CPU-based counterparts, and our GPU joins with optimizations on hardware features are another 2-5 times faster than the basic GPU implementation.

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. For instance, the join algorithms presented in our paper mainly deal with data parallelism in a single query operator. Our ongoing work includes designing and implementing other GPU data manipulation primitives and algorithms for a complete set of relational query operators on the new GPU architecture. With these primitives and operators, we will not only be able to handle intra-operator data parallelism but also inter-operator parallelism such as pipelining. Finally, it is an interesting future direction to extend GPU query processing to disk-based data that may or may not already reside inside the GPU video memory.

8. REFERENCES

[1] Intel compiler 9.1 documentation, ftp://download.intel.com/support/performancetools/c/windows/v9/icl_doc.zip.

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

[3] A. Ailamaki, N. Govindaraju, S. Harizopoulos and D. Manocha. Query co-processing on commodity processors (Tutorial). VLDB, 2006.

[4] N. Bandi, C. Sun, D. Agrawal and A. El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. VLDB, 2004.

[5] K. E. Batcher. Sorting networks and their applications. AFIPS Spring Joint Computer Conference, 1968.

[6] G. E. Blelloch. Prefix sums and their applications. Technical report, CMU-CS-90-190, Nov 1990.

[7] P. Boncz, S. Manegold and M. Kersten. Database architecture optimized for the new bottleneck: memory access. VLDB, 1999.

[8] J. Cieslewicz, J. Berry, B. Hendrickson and K. A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. DaMoN 2006.

[9] 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.

[10] B. Gold, A. Ailamaki, L. Huston and B. Falsafi. Accelerating database operators using a network processor. DaMoN, 2005.

[11] N. Govindaraju, J. Gray, R. Kumar and D. Manocha. GPUTeraSort: high performance graphics coprocessor sorting for large database management. SIGMOD, 2006.

[12] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. SIGMOD, 2004.

[13] N. Govindaraju, N. Raghuvanshi and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. SIGMOD, 2005.

[14] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: limitations and opportunities. CIDR, 2007.

[15] S. Harizopoulos, V. Liang, D. Abadi and S. Madden. Performance tradeoffs in read-optimized databases. VLDB, 2006.

[16] D. Horn. Lib GPU FFT, http://sourceforge.net/projects/gpufft/. 2006.

[17] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. VLDB, 1991.

[18] A. Lamarca and R. Ladner. The influence of caches on the performance of sorting. SODA, 1997.

[19] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. Supercomputing, 2001.

[20] B. Liu and E. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. VLDB, 2005.

[21] H. Lu, K. Tan and M. Shan. Hash-based join algorithms for multiprocessor computers with shared memory. VLDB, 1990.

[22] 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, Volume 26, 2007.

[23] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. VLDB, 1999.

[24] D. A. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD, 1989.

[25] A. Shatdal, C. Kant and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB, 1994.

[26] 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.

[27] C. Sun, D. Agrawal and A. El Abbadi. Hardware acceleration for spatial selections and joins. SIGMOD, 2003.

[28] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. Supercomputing, 1991.