

June 2020

## Relational Joins on GPUs for In-Memory Database Query Processing

Ran Rui  
*University of South Florida*

Follow this and additional works at: <https://digitalcommons.usf.edu/etd>



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

---

### Scholar Commons Citation

Rui, Ran, "Relational Joins on GPUs for In-Memory Database Query Processing" (2020). *USF Tampa Graduate Theses and Dissertations*.  
<https://digitalcommons.usf.edu/etd/8484>

This Dissertation is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact [digitalcommons@usf.edu](mailto:digitalcommons@usf.edu).

# Relational Joins on GPUs for In-Memory Database Query Processing

by

Ran Rui

A dissertation submitted in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy in Computer Science and Engineering  
Department of Computer Science and Engineering  
College of Engineering  
University of South Florida

Major Professor: Yi-Cheng Tu, Ph.D.  
Mehran Mozaffari Kermani, Ph.D.  
Yu Sun, Ph.D.  
Andres Tejada-Martinez, Ph.D.  
Sameer Varma, Ph.D.

Date of Approval:  
June 23, 2020

Keywords: Parallel Computing, Algorithm Design, Query Performance, Emerging  
Hardware, Algorithm Optimization

Copyright © 2020, Ran Rui

## **Dedication**

To my dear wife Wen, and all my families and friends who have supported me throughout this journey.

## Table of Contents

|  |     |
|--|-----|
| List of Tables   | iv  |
| List of Figures  | v   |
| Abstract   | vii |
| Chapter 1: Introduction  | 1   |
| Chapter 2: Background  | 3   |
| 2.1 Overview   | 3   |
| 2.2 Related Work   | 4   |
| Chapter 3: Join Algorithms on GPUs: A Revisit                              | 9   |
| 3.1 Experimental Setup   | 12  |
| 3.2 Main Results   | 13  |
| 3.2.1 GPU Architecture   | 13  |
| 3.2.2 Performance Comparison   | 16  |
| 3.2.2.1 Code Scalability   | 17  |
| 3.2.2.2 Time Breakdown   | 18  |
| 3.3 Optimization on New GPU Architecture                                   | 19  |
| 3.3.1 Cache/Register Optimization  | 20  |
| 3.3.2 Performance Evaluation   | 21  |
| 3.4 Other Considerations   | 23  |
| 3.4.1 Energy / Power Consumption   | 24  |
| 3.4.2 Floating Point Performance   | 25  |
| 3.4.3 Limitation of Memory Size  | 26  |
| 3.5 Discussions  | 27  |
| 3.5.1 Main Findings and Recommendations                                    | 28  |
| 3.5.2 Response to Concerns in [1]  | 29  |
| Chapter 4: Fast In-Core Join Algorithms on GPUs: Design and Implementation | 31  |
| 4.1 Join Algorithm Design on GPUs  | 34  |
| 4.1.1 GPU Architecture   | 34  |
| 4.1.1.1 New Features of GPUs   | 35  |
| 4.1.2 Hash Join  | 36  |

|            |   |    |
|------------|---|----|
| 4.1.2.1    | The Partitioning Stage  | 37 |
| 4.1.2.2    | The Probe Stage   | 39 |
| 4.1.2.3    | Skew Handling   | 42 |
| 4.1.3      | Sort-Merge Join   | 43 |
| 4.1.3.1    | The Sort Stage  | 43 |
| 4.1.3.2    | The Merge Join Stage  | 45 |
| 4.1.4      | Handling Large Input Tables                                   | 47 |
| 4.2        | Evaluations   | 49 |
| 4.2.1      | Experimental Setup  | 49 |
| 4.2.2      | Experimental Results  | 50 |
| 4.2.2.1    | Comparing with Existing GPU Code                              | 50 |
| 4.2.2.2    | Comparing with Latest CPU Code                                | 54 |
| 4.2.2.3    | Time Breakdown  | 55 |
| 4.2.2.4    | Effects of Join Selectivity                                   | 57 |
| 4.2.2.5    | Effects of Direct Output                                      | 57 |
| 4.2.2.6    | Effects of Skewed Data  | 60 |
| 4.2.2.7    | Joins under Large Data  | 61 |
| Chapter 5: | Efficient Join Algorithms For Large Tables with Multiple GPUs | 63 |
| 5.1        | Challenges in Multi-GPU Joins                                 | 65 |
| 5.1.1      | Limited PCI-E Bandwidth                                       | 66 |
| 5.1.2      | Complex Inter-GPU Communication Pattern                       | 67 |
| 5.2        | Large Table Join Algorithms                                   | 68 |
| 5.2.1      | Block-Based Nested-Loop Join                                  | 68 |
| 5.2.1.1    | Performance Analysis  | 70 |
| 5.2.2      | Global Sort-Merge Join  | 72 |
| 5.2.2.1    | Sorted Run Generation   | 73 |
| 5.2.2.2    | Multi-GPU Merge of Sorted Runs                                | 74 |
| 5.2.2.3    | Multi-GPU Merge Join  | 75 |
| 5.2.2.4    | Performance Analysis  | 76 |
| 5.2.3      | Hybrid Join   | 76 |
| 5.2.3.1    | Multi-GPU Radix Partition                                     | 78 |
| 5.2.3.2    | In-Core Sort-Merge Join                                       | 81 |
| 5.2.3.3    | Performance Analysis  | 82 |
| 5.3        | Experiments   | 83 |
| 5.3.1      | Total Running Time  | 85 |
| 5.3.1.1    | Scalability   | 86 |
| 5.3.1.2    | Running Time Breakdown  | 88 |
| 5.3.1.3    | The Effects of CUDA Stream Pipeline                           | 90 |
| 5.3.2      | Effects of Other Factors                                      | 90 |
| 5.3.2.1    | $ R  :  S $ Ratio   | 90 |
| 5.3.2.2    | Selectivity   | 92 |

|                                   |  |     |
|-----------------------------------|--|-----|
| 5.3.2.3                           | Data Skewness  | 92  |
| 5.3.3                             | Speedup Over The State-of-The-Art CPU and GPU Algorithms | 95  |
| 5.3.4                             | Discussions  | 96  |
| Chapter 6: Conclusions            |  | 98  |
| References                        |  | 101 |
| Appendix A: Copyright Permissions |  | 107 |

## List of Tables

|           |  |    |
|-----------|--|----|
| Table 3.1 | Specifications of hardware mentioned in this chapter   | 10 |
| Table 3.2 | Performance of four join algorithms on different GPUs and CPUs                               | 15 |
| Table 3.3 | NINLJ performance on GTX Titan under read-only and L2 cache optimizations                    | 23 |
| Table 3.4 | Running time (ms) of the prefix scan kernel optimized by shuffle instruction                 | 23 |
| Table 3.5 | Average active power consumption (watt)  | 25 |
| Table 4.1 | Specifications of hardware mentioned in this chapter   | 49 |
| Table 4.2 | Resource utilization of major kernels in the new and old GPU sort-merge join code            | 51 |
| Table 4.3 | Resource utilization of major kernels in the new and old GPU hash join code                  | 52 |
| Table 5.1 | Common symbol definitions  | 68 |
| Table 5.2 | Speedup of multi-GPU runs of different algorithms over single-GPU runs on GTX Titan X pascal | 84 |
| Table 5.3 | Speedup of multi-GPU runs of different algorithms over single-GPU runs on Tesla P100         | 84 |

## List of Figures

|            |   |    |
|------------|---|----|
| Figure 3.1 | Memory hierarchy in Kepler architecture   | 14 |
| Figure 3.2 | Relative capacity of hardware resources and join performance between new and old CPUs/GPUs            | 14 |
| Figure 3.3 | Time spent on data transmission and join processing   | 19 |
| Figure 3.4 | Data movement in the modified NINLJ algorithm   | 20 |
| Figure 3.5 | Data access pattern using shuffle instructions  | 22 |
| Figure 3.6 | Energy consumption of different CPUs/GPUs in processing four join algorithms                          | 24 |
| Figure 3.7 | Relative performance of NINLJ with SP/DP keys in different CPUs/GPUs under two table sizes            | 27 |
| Figure 4.1 | Layout of latest NVidia GPU architecture  | 34 |
| Figure 4.2 | Shared histogram used in Partitioning and Reordering in GPU hash join                                 | 37 |
| Figure 4.3 | Workflow of threads of probe stage in hash join   | 40 |
| Figure 4.4 | A case of direct output buffer for GPU hash join, showing Thread 3 acquiring chunk 4 as output buffer | 42 |
| Figure 4.5 | Parallel merge with 7 threads using Merge Path  | 44 |
| Figure 4.6 | Overlapping data transmission and join processing using two CUDA streams                              | 48 |
| Figure 4.7 | Speedup of new GPU join algorithms over existing GPU code under different table sizes                 | 53 |
| Figure 4.8 | Speedup of our GPU code over the latest CPU code  | 54 |
| Figure 4.9 | Execution time breakdown (percentage) of new and old GPU algorithms running on Titan X                | 56 |



|             |  |    |
|-------------|--|----|
| Figure 4.10 | Impact of join selectivity on speedup of Titan X over E5-2630v3 under data size 64M          | 56 |
| Figure 4.11 | Speedup of direct output vs. double probe in the new hash join                               | 58 |
| Figure 4.12 | Performance gained by using shared memory buffer pointer vs. global buffer pointer           | 59 |
| Figure 4.13 | Performance of CPU/GPU code under different levels of data skewness                          | 60 |
| Figure 4.14 | Speedup of Titan X over E5-2630v3 with large tables  | 61 |
| Figure 5.1  | A typical system layout of four GPUs connecting to one CPU socket                            | 66 |
| Figure 5.2  | Overview of block-based nested-loop join   | 69 |
| Figure 5.3  | Loading of inner table chunks and GPU peer-to-peer data transmission in the nested-loop join | 71 |
| Figure 5.4  | Global sorting with two GPUs   | 72 |
| Figure 5.5  | Radix partition with two GPUs  | 79 |
| Figure 5.6  | Running time of three join algorithms with PCIE and NVLink                                   | 83 |
| Figure 5.7  | Running time breakdown of our join algorithms under 8-billion records and 4 GPUs             | 88 |
| Figure 5.8  | Data transfer time of the three join algorithms  | 89 |
| Figure 5.9  | Speedup of 3-stream pipeline   | 91 |
| Figure 5.10 | Running time of various $ R  :  S $ ratio  | 93 |
| Figure 5.11 | Running time of various selectivity  | 93 |
| Figure 5.12 | Running time of three GPU join algorithms against skewed data                                | 93 |
| Figure 5.13 | The speedup of our out-of-core GPU join algorithms over a CPU-based hash join algorithm      | 94 |

## Abstract

Relational join processing is one of the core functionalities in database management systems. Implementing join algorithms on parallel platforms, especially modern GPUs, has gain a lot of momentum in the past decade. This dissertation addresses the following issues on GPU join algorithms. First, we present empirical evaluations of a state-of-the-art work on GPU-based join processing. Since 2008, the compute capabilities of GPUs have increased following a pace faster than that of the multi-core CPUs. We run a comprehensive set of experiments to study how join operations can benefit from such rapid expansion of GPU capabilities. We also present improved GPU programs that take advantage of new GPU hardware/software features such as read-only data cache, large L2 cache, and shuffle instructions. Second, we report new design and implementation of join algorithms with high performance under today's GPGPU environment. In particular, we overhaul the popular radix hash join and redesign sort-merge join algorithms on GPUs by applying a series of novel techniques to utilize the hardware capacity of latest Nvidia GPU architecture and new features of the CUDA programming framework. Our algorithms take advantage of revised hardware arrangement, larger register file and shared memory, native atomic operation, dynamic parallelism, and CUDA Streams. Lastly, we explore how join processing would benefit from the adaptation of multiple GPUs. We identify the low rate and complex patterns of data transfer among the CPU and GPUs as the main challenges in designing efficient algorithms for large table joins, and we propose three distinctive designs of multi-GPU join algorithms, namely, the nested loop, global sort-merge and hybrid joins to overcome such challenges. Extensive experiments running on multiple databases and two different

hardware configurations demonstrate high scalability of our algorithms over data size and significant performance boost brought by the use of multiple GPUs. Furthermore, our algorithms achieve much better performance as compared to existing join algorithms, with a speedup up to 27.5X and 2.9X over best known code developed for multi-core CPUs and GPUs respectively.

## Chapter 1: Introduction

The join operation is an important and widely used function in relational database management systems. It finds matching tuples from two tables which share some common attributes. To accomplish the task, a simple join algorithm exhaustively searches all the possible tuple pair, creating a quadratic searching space relative to the table size. In order to reduce the cost of joins, we can use auxiliary data structures (i.e. index and hash table) or sort (either partially or fully) the input tables so that number of tuple pairs to be compared is significantly smaller. The join operation can either be a stand-alone query or be the building block of more complicated queries.

As we have entered the “big data” era, data are generated at an unprecedentedly fast rate. Therefore, processing the increasing volume of data also requires faster consumption speed, otherwise the data would be piled up and never be processed in time. The same situation applies for the relational join processing. Therefore, faster hardware and algorithms are urgently needed for joins on large data.

Among all the hardware platforms, Graphics Processing Units(GPUs) becomes more attractive due to its computing power and price efficiency. A high-end GPU’s computing capability is equivalent to a small-scale CPU-based clusters that have up to tens of nodes. However, designing join algorithms on GPUs is not trivial and requires delicate efforts.

In this dissertation, we address the issues of design and implementation of GPU-based join algorithms that are capable of processing large data. Our contribution consists of three aspects. First, by running existing GPU-based join algorithms on modern GPUs, we find that the existing algorithms that were designed for older GPU architectures cannot fully

utilize the hardware resources of the latest GPU architecture. Applying cache optimizations easily improves the algorithms' performance by 30%-50%. However, a brand new algorithmic design is needed for more improvement. Second, we propose new GPU-based sort-merge join and hash join algorithms that are designed with the new GPU architecture and new hardware/software features in mind. They are optimized to fully utilized the GPU's computing resources. Comparing with previous GPU join algorithms, our code achieves a large speedup, and the utilization of GPU resources increases considerably. Third, we enable large table join processing on GPUs by taking advantage of multi-GPU environments. We identify the major bottleneck in a multi-GPU system: I/O overhead, and propose three distinct algorithms that work with different join conditions to minimize such overhead. The best of our algorithms shows linear scalability on table size, and also achieved much better performance with the use of multiple GPUs.

This dissertation is organized as follow. In Chapter 2, we briefly introduce background knowledge and summarize representative existing work on parallel join algorithms. In Chapter 3, we present the experimental result of older GPU join algorithms running on modern GPUs. In Chapter 4, we present our new design of GPU-based join algorithms. In Chapter 5, we address the issues of join algorithms in a multi-GPU environment. Chapter 6 gives conclusions of the whole research.

## Chapter 2: Background

### 2.1 Overview

The join operation takes two tables/relations as the input. Each table contains one or more attributes/columns. Each row of the table representing a single record is called a tuple. When the common attributes of two tuples from the two input tables satisfy the join condition (e.g. equality or range), this pair of tuples are put into the output result. For simplicity, we use join to refer to equality join in this dissertation.

There are three major variants of join implementation based on their procedural differences. We assume that there are two input tables A and B, the number of tuples of which are M and N respectively. The most straightforward method is called nested-loop join which loops over all the pairs of tuples from the two input tables. It results in a total time complexity of  $O(M * N)$ . Several techniques can be used to improve the nested-loop join. One of them is using blocking mechanism that reading one or multiple blocks of data rather than fetching a single tuple during each iteration. By doing that we can save considerable amount of I/O cost. Another commonly used nested loop join improvement is taking advantage of the index structure already built for at least one of the tables. The assumption of using indexed nested loop join is that the index must be built on the join key, or the attribute of the join condition. In this case, we only need to traverse one table and search the matching key into the other table using index. The two optimization techniques can be used together to further improve the running time of nested loop join to linear time, excluding the cost of building the index.

The second variant is called sort-merge join. The basic idea is to sort the two tables based on the join key, and then using merge join which resembles the merge sort to get the matching tuples. The total cost of sort-merge join depends on the sorting algorithms and the merge join design. A typical sort-merge join takes  $O(M * \log(M) + N * \log(N) + M + N)$  running time.

The third variant of relational join algorithm is hash join. First, a hash-based index is built on the join key using a hash function  $h$  for one of the tables. Then the other table is traversed and for each tuple, compute its hash value using hash function  $h$ , then search for matching tuples in the index of the first table. A well-designed hash join takes  $O(M + N)$  running time.

## 2.2 Related Work

GPGPU has become very popular high-performance computing technique in the last few years. The SIMD architecture of GPU provides tremendous amount of computing power under very high energy efficiency – nearly 30% of the Top500 supercomputers in the world has deployed GPUs in their architecture [2].

Before the emergence of GPGPU programming languages such as CUDA and OpenCL, there were already a number of studies that used GPUs to accelerate database operations via graphic APIs. Sun *et al.* [3] utilized the rendering and searching functions of GPU to speed up spatial database selections and joins. Their hardware-assisted method reached a speed-up of 4.8-5.9X in joins comparing to CPUs. In a later work, Bandi *et al.* [4] extended that proposal to a practical scenario by integrating GPU-assisted spatial operations into a commercial DBMS. Govindaraju *et al.* [5] proposed a set of commonly used operations including selections, aggregations and semi-linear queries implemented on GPUs. The same group implemented a high performance bitonic sorting algorithm on GPUs that served as an essential part of many other database operations [6]. However, the studies mentioned above

were all based on very old GPU architectures, which were not optimized for general-purpose computing. They also had to rely on graphic APIs such as OpenGL and DirectX, which limited the programmability and functionality of their implementations.

Since the major GPU manufacturers evolved their products to adopting the Unified Shading Architecture around 2007 [7], there has been unprecedented effort devoted to the GPGPU paradigm, especially after the release of advanced GPU computing models such as CUDA [8] and OpenCL [9]. The same trend has also affected the database community. He *et al.* [10] proposed very efficient gather and scatter operations on CUDA-enabled GPUs. These algorithms made full use of the high memory bandwidth of GPUs by addressing computation for coalesced memory access, thus eliminating the costly overhead of random memory access. They also developed plausible solutions for data read and write primitives of database operations on GPUs. Based on that, He *et al.* [1] developed a comprehensive package of GPU-based database algorithms including a series of primitives and four join algorithms developed on top of those primitives.

With the computing power of a first generation CUDA-supported GPU, the primitives reached speedup of 2.4-27.3X while the four join algorithms achieved 1.9-7.0X speedup compared to a quad-core Intel CPU. In an extended version [11] of [1], the same team studied performance modeling and combining CPUs and GPUs for relational data processing. Since the core issue we are interested in is GPU performance, we will only refer to [1] for comparison and discussions in this dissertation. In [12], Kaldewey *et al.* used Unified Virtual Addressing (UVA) to alleviate the difficulty of explicitly copying data to GPUs by enabling the GPU accessing host memory directly. Bakkum *et al.* [13] integrated a GPU-accelerated SQL command processor into the open-source SQLite system. Specifically, the command processor boosted the performance of SQL SELECT queries in the database system, where 20-70X speedups were achieved. Due to the limitation of SQLite, this result was achieved by comparing with single-thread CPU implementation. However, our work is based on a



multi-core, multi-thread enabled code which make full use of the maximum performance of recent hardware platforms. Apart from pure GPU-based studies, there were also studies on further improving the overall system performance via distributing computation to both CPU and GPU [11, 14].

Designing and optimizing algorithms for join and other database operators on many/multi-core systems has been an active topic in the database field. The focus on the CPU-based algorithms are to exploit data level and task level parallelism. SIMD instructions such as SSE and AVX on Intel processors are often used for data level parallelism. Kim *et al.* proposed sort-merge join and hash join algorithms with SIMD optimizations on a Core i7 system [15]. By comparing the two algorithms, the authors concluded that the hash join is faster and a wider SIMD instruction could benefit sort-merge join. In [16], by studying various hash join implementations, the authors found that a simple non-partition hash join with shared hash table outperformed other more complex and hardware-conscious implementations. However, this result was based on a particular dataset. In [17], Balkesen *et al.* drew an opposite conclusion. They claimed that hardware-conscious optimization is still required to achieve optimal performance in hash join. Their radix hash join implementation with the bucket chain method proposed by Manegold *et al.* [19] is the fastest. In [20], Balkesen *et al.* revisited the classic sort-merge join vs. hash join topic with comprehensive experiments and analysis. They provided the fastest implementation of both algorithms, and claimed that in most cases the hash join outperforms sort-merge join. The sort-merge join was only able to narrow the gap when the data is very large. To resolve the high memory consumption of hash join, Barber *et al.* proposed a memory-efficient hash join by using a concise hash table while not sacrificing performance [21]. Albutiu *et al.* proposed a parallel sort-merge join in which each thread works on its local sorted runs in a NUMA environment to avoid expensive cross-region communication [18]. To deal with the high memory consumption of

hash join, Barber *et al.* proposed a memory-efficient hash join by using a concise hash table while maintaining competitive overall performance [21].

On the GPU side, He *et al.* designed a series of GPU-based data operators as well as four join algorithms [1]. Their algorithms were designed to take advantage of an early generation of CUDA-enabled GPUs. Bakkum *et al.* implemented an SQL command processor that was integrated into an open-source database software [13]. Yuan *et al.* studied the performance of GPUs for data warehouse queries and provided insights of narrowing the gap between the computing speed and data transfer speed [22]. Wu *et al.* proposed an implementation of compiler and operators for GPU-based query processing [23]. Kaldewey *et al.* revisited the join processing on GPU to utilize the Unified Virtual Addressing (UVA) to alleviate the cost of data transfer [12]. There are also reports of CPUs' working cooperatively with GPUs to process data [11, 14]. Close in spirit to [13, 22, 23], we are in the process of developing a scientific data management system named G-SDMS that features a push-based I/O mechanism and GPU kernels for data processing. A sketch of the G-SDMS design can be found in [24].

There are controversial views on whether GPU is superior to CPU in join processing. In [1], the authors claimed a 2-7X GPU-to-CPU speedup for various join algorithms. However, in [15], more optimized CPU code achieved up to 8X speedup over GPU joins. By studying various operators on CPUs and GPUs, Lee *et al.* claimed that GPU is about 2.5X as efficient as CPU on average [25]. Our previous work [26] showed that hardware development over the past few years affects both CPU and GPU joins. By testing the same CPU and GPU code used in [1], it is shown that the GPUs were up to 19X faster in sort-merge join and 14X faster in hash join. However, such experiments did not consider the most recent development of CPU and GPU joins. In this dissertation, we propose join algorithms that are optimized for the latest GPUs, and compare their performance with the best CPU code presented in [17] and [20].

However, current studies rarely address the issue of large table join using multiple GPUs. In [12], a proposal was to use UVA for GPU join processing with early generations of GPUs. The UVA in CUDA has since been enhanced with hardware page-fault and software prefetching support, and renamed as Unified Memory (UM). More recent work studied the performance of using UM in out-of-core join processing on GPUs [28, 29]. However, they found that the throughput achieved by UM is many times lower than a carefully designed data movement strategy. To the best of our knowledge, no studies have addressed the use of multiple GPUs to improve join performance.

### Chapter 3: Join Algorithms on GPUs: A Revisit

<sup>1</sup> Many-core architectures such as Graphics Processing Units (GPU) have become a popular choice of high-performance computing (HPC) platform. A modern GPU chip consists of thousands of cores that deliver tremendous computing power. It is also equipped with high speed memory modules to satisfy the data communication needs of the cores. Such characteristics of GPUs, along with the general-purpose programming frameworks such as Compute Unified Device Architecture (CUDA) [8] and Open Computing Language (OpenCL) [9], have drawn much attention from the HPC communities.

The database community is also among those who benefited from general-purpose GPU (GPGPU) computing technology. In recent years, a number of studies have provided evidence of GPU's capability to speed up database operations [30, 11, 1, 23, 12, 13, 31, 22, 24]. In relational DBMSs, the most time-consuming operation is join. In 2008, He *et al.* published their work in the design and implementation of four major join algorithms on GPUs [1]: block-based non-indexed nested loop join (NINLJ), indexed nested loop join (INLJ), sort merge join (SMJ), and radix hash join (HJ). They thoroughly compared the performance of these algorithms on a mainstream GPU device with that of a highly-optimized CPU version and demonstrated that GPU achieved up to a 7X speedup over CPU, which is a significant improvement by any standards. In this chapter, we report the results of a comprehensive set of experiments running the program developed by He *et al.*. However, our study serves more

---

<sup>1</sup>This chapter was published in 2015 IEEE International Conference on Big Data, pp. 2541-2550, doi: 10.1109/BigData.2015.7364051. Permission is included in Appendix A

significant purposes than simply verifying the findings of [1]. Instead, we aim at drawing an up-to-date and panoramic image of GPGPU as a means for processing join operators.

**Table 3.1:** Specifications of hardware mentioned in this chapter

| Device           | CPU                        |                              |                            | GPU                                   |                                    |                           |
|------------------|----------------------------|------------------------------|----------------------------|---------------------------------------|------------------------------------|---------------------------|
|                  | Xeon<br>E5-2640<br>V2      | Core i7<br>3930K             | Core 2<br>Quad<br>Q6600    | GTX Titan                             | GTX 980                            | 8800 GTX                  |
| Date released    | Q3 2013                    | Q4 2011                      | Q1 2007                    | Q1 2013                               | Q3 2014                            | Q4 2006                   |
| Core Speed       | 2.00GHz                    | 3.20GHz                      | 2.40GHz                    | 0.84GHz                               | 1.13GHz                            | 0.58GHz                   |
| Core Count       | 8                          | 6                            | 4                          | $14 \times 192$                       | $16 \times 128$                    | $8 \times 32$             |
| Cache Size       | L1:<br>512KB<br>L2:<br>2MB | L1:<br>384KB<br>L2:<br>1.5MB | L1:<br>256KB<br>L2:<br>8MB | L1:<br>64KB $\times$ 14<br>L2: 1536KB | L1:<br>96KB $\times$ 16<br>L2: 2MB | L1:<br>16KB $\times$ 8    |
| RAM              | DDR3 Triple Channel        |                              | DDR2<br>Dual<br>Channel    | GDDR5<br>6GB<br>384 bit               | GDDR5<br>4GB 256<br>bit            | GDDR3<br>768MB<br>384 bit |
| Memory Bandwidth | 38.4GB/s                   |                              | 12.8GB/s                   | 288GB/s                               | 224GB/s                            | 86.4GB/s                  |
| Max GFLOPS       | 128                        | 153.6                        | 38.4                       | 4494                                  | 4612                               | 345.6                     |
| Max TDP          | 95W                        | 130W                         | 105W                       | 250W                                  | 230W                               | 155W                      |
| Launch Price     | 889 USD                    | 594 USD                      | 530 USD                    | 999 USD                               | 549 USD                            | 599 USD                   |

With the promising performance shown in existing work, it is worth exploring the actual benefit of using GPUs for processing database operators as of today. This is especially important in that the GPU industry has since released new devices that carry many times of computing capabilities as those found in 2008. For example, Table 3.1 shows the specifications of several Nvidia GPUs, including the 8800 GTX that is used as the testbed in [1]), and

the GTX Titan plus GTX 980 that we use in our study.<sup>2</sup> We can easily see that the memory bandwidth of the GTX Titan is 3.3 times as that of the 8800 GTX, and the raw computing power is 13 times as high. It would be interesting to see how such increase of computing capabilities is reflected in performing database operations. Therefore, an important objective of our work is to empirically evaluate the performance of the aforementioned GPU join algorithms in today’s GPU devices. To that end, we run the code used in [1] in modern GPUs and CPUs and compare their performance. In particular, the code includes both GPU and CPU versions of four join algorithms mentioned above: NINLJ, INLJ, SMJ, and HJ, as well as a set of data primitives such as map, sort, and prefix-scan. Our experiments show that, by calculating the end-to-end running time, the GPUs achieve up to 20X speedup over the CPUs in the four join algorithms. It is clear that the performance gap between GPU and CPU in join processing is widened since 2008. The second objective is to evaluate the full potential of GPGPU in processing joins by considering the many new techniques implemented in GPUs in recent few years. Specifically, we redesign some of the aforementioned GPU programs by taking advantage of new hardware and software features such as read-only data cache, large L2 cache, and shuffle instructions. By applying such optimizations, extra performance improvement of 30-52% is observed in various kernels. Finally, we evaluate the join programs from a few other perspectives such as energy efficiency, floating-point performance, and data size considerations. Those are done in response to relevant discussions presented in [1] and further reveal the advantages and limitations of GPGPU from a database perspective. In short, we find that today’s GPUs are significantly faster on floating point operations, can process more on-board data, and achieves higher energy efficiency than modern CPUs. The availability of new tools has made program development and optimization on GPUs much easier than before.

---

<sup>2</sup>Information is mainly extracted from the Intel and Nvidia corporate websites, with other information obtained from [www.techpowerup.com](http://www.techpowerup.com)

### 3.1 Experimental Setup

Our testbed is a high-end workstation featuring 48GB of DDR3 memory and one 512GB SSD disk. The motherboard is an AsRock X79 Extreme 11 hosting seven PCI-E 3.0 slots with full 16X speed and can support up to four double-width GPU cards. Note that each PCI-E slot provides approximately 15.8GB/s of bandwidth [32] for efficient data transfer between the host and the GPU.

We obtain the entire code package introduced in [1] from its first author, Dr. Bingsheng He. This package includes both CPU and GPU versions of four join algorithms and five join-related data primitives. We test the code with a variety of CPUs and GPUs. However, in this chapter we only report the results of two GPUs - the Nvidia Geforce GTX 980 and the Nvidia GTX Titan, in comparison to two CPUs: Intel Core i7-3930K and Intel Xeon E5-2640v2. The specifications of the chosen hardware are shown in Table 3.1. Based on their prices, the Core i7 and GTX 980 are mid-range hardware found in typical desktop computers while the Xeon E5 and GTX Titan represent those found in powerful workstations. Note that the CPU and GPU within each group are at the same price range – this allows a fair comparison in terms of cost efficiency. We also tested other GPU products such as the Nvidia Tesla K20 and K40 [33]. However, these devices are way more expensive than (yet with only comparable performance as) the Titan therefore we skip the discussions on such results in this chapter. Interested readers can refer to a longer version of this chapter [34] for such details.

Our workstation runs Windows 7 (SP1) with Visual Studio 2010 as the program development environment. For GPU computing, we use CUDA 6.0 to compile the GTX Titan code and CUDA 6.5 for the GTX980. The code was compiled and tested with the best configuration and parameters discussed in the previous work [1]. As in [1], each tuple in the database table contains an *id* and a *key value*. Unless specified otherwise, the *key values* are

integers ranging from 0 to  $2^{30}$ . Such values are generated randomly, and an ID is specified to each key value in order. As in the original code, we fixed the number of output tuples in our experiments by setting the tuple matching rate between two tables to 0.1% . The number of output tuples is changed only in one experiment for the purpose of testing the effects of such changes on the overall performance. In all experiments, both the inner and outer tables are of the same size.

Performance measurement is done by the built-in timing functions in the original code for both CPUs and GPUs. To measure the power consumption of hardware, we connect a WattsUp Pro power meter [35] to our machine. A software reads the power consumption and power readings are sent to the computer from the power meter via a USB connection. Energy consumption is obtained by integrating all the runtime power readings under the assumption that power does not change within the sampling window.

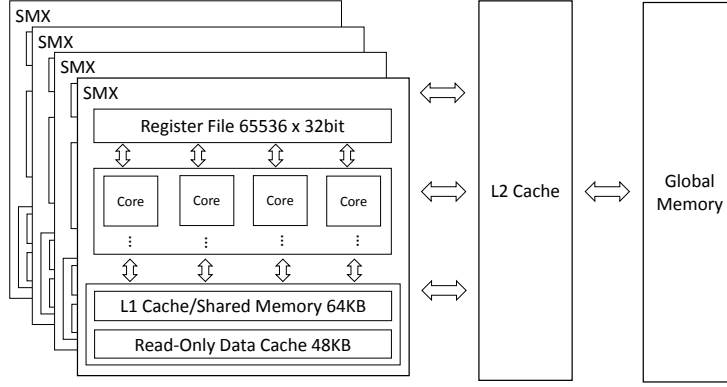
## 3.2 Main Results

In this section, we report the performance of the original code provided by He *et al.* for both CPUs and GPUs. We focus on performance comparison between GPUs and CPUs found in today’s market. As mentioned earlier, this gives an overview of the advantages of GPUs for processing joins over CPUs, and whether such advantages increase/decrease over time.

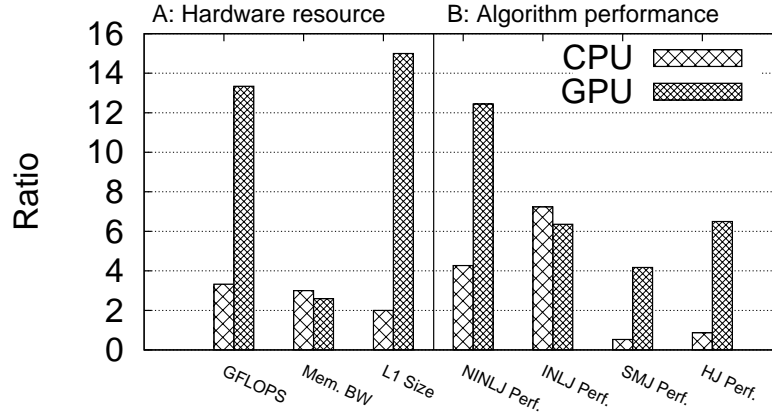
### 3.2.1 GPU Architecture

Before starting our discussions on GPU-based joins, we need a close look at the typical GPU architecture. Take the GTX Titan’s Kepler architecture as an example (Figure 3.1): it consists of a few Streaming Multiprocessors (SMX), each of which is regarded as a fully functional computing unit. Within an SMX, there are many (e.g., 192 in Kepler) computing cores, certain amount of cache, and a considerably large register file. The register





**Figure 3.1:** Memory hierarchy in Kepler architecture



**Figure 3.2:** Relative capacity of hardware resources and join performance between new and old CPUs/GPUs

pool consists of tens of thousands of 32-bit registers providing sufficient private storage for threads. Each SMX has its own L1 cache for fast data access and synchronization among threads. A unique feature of Nvidia GPUs is: part of the L1 cache can be configured to be a programmable section called *shared memory* (SM). Similar to traditional CPU architectures, GPUs have a multi-level memory system: in addition to the L1 cache inside the SMX, there are also L2 cache and the *global memory* (GM) shared by all SMXs. The global memory, being the main data storage unit for GPUs, often comes with a size of a few GBs and high bandwidth following the GDDR5 standard.

**Table 3.2:** Performance of four join algorithms on different GPUs and CPUs

| Algorithm | Data Size | Running time (second) |          |           |        | GPU to CPU Speedup |          |           |
|-----------|-----------|-----------------------|----------|-----------|--------|--------------------|----------|-----------|
|           |           | E5-2640               | i7-3930K | GTX Titan | GTX980 | GTX980/E5          | Titan/i7 | Base-line |
| NINLJ     | 1M        | 123.74                | 109.36   | 14.74     | 6.03   | 20.51              | 7.42     | 7.0       |
|           | 2M        | 492.99                | 434.17   | 58.66     | 24.25  | 20.33              | 7.40     | —         |
|           | 4M        | 1967.14               | 1719.55  | 235.48    | 97.18  | 20.24              | 7.30     | —         |
|           | 8M        | 7823.65               | 6846.33  | 957.01    | 388.90 | 20.12              | 7.15     | —         |
| INLJ      | 16M       | 0.58                  | 0.45     | 0.11      | 0.11   | 5.47               | 4.09     | 6.1       |
|           | 32M       | 1.28                  | 0.99     | 0.24      | 0.20   | 6.53               | 4.13     | —         |
|           | 64M       | 2.97                  | 2.25     | 0.55      | 0.46   | 6.43               | 4.09     | —         |
|           | 128M      | 5.93                  | 5.03     | 1.24      | 1.07   | 5.55               | 4.06     | —         |
| SMJ       | 16M       | 9.41                  | 6.86     | 0.45      | 0.48   | 19.73              | 15.24    | 2.4       |
|           | 32M       | 18.32                 | 12.48    | 0.97      | 1.04   | 17.70              | 12.87    | —         |
|           | 64M       | 36.02                 | 24.82    | 2.09      | 2.24   | 16.11              | 11.88    | —         |
| HJ        | 16M       | 2.87                  | 2.04     | 0.36      | 0.20   | 14.28              | 5.67     | 1.9       |
|           | 32M       | 5.77                  | 4.08     | 3.55      | 3.33   | 1.73               | 1.15     | —         |
|           | 64M       | 11.66                 | 8.27     | 4.15      | 3.70   | 3.15               | 1.99     | —         |
|           | 128M      | 24.68                 | 17.15    | 5.37      | —      | —                  | 3.19     | —         |

Apart from increasing computing resources, GPUs have better power/energy efficiency as well. Although the scale of the GPU chips has increased due to the larger number of cores, their power consumption (indicated by TDP - thermal design power) remains at the same level, which implies better energy efficiency than CPUs when performance is considered.

### 3.2.2 Performance Comparison

Table 2 shows the performance of the four join algorithms on the two CPUs and two GPUs mentioned above. Each data point is the average of four runs with identical setups.<sup>3</sup> The data size is presented in number of tuples (each tuple is 8 bytes long) and both tables in a join are of the same size.

Our first observation is from the CPU side: the 6-core i7-3930K has better performance than the 8-core Xeon E5 in all experiments, although the latter is a newer CPU with a higher price tag. We believe the high clock speed of the i7-3930K compensates for the smaller number of cores. This also reflects a general trend of modern CPU design: the focus moved from computing performance to other factors such as energy efficiency. We have similar observations from the GPUs: the less expensive GTX 980 outperforms the high-end Titan in all but the SMJ experiments. This is not really a surprise to us: the main selling point for the Maxwell architecture is higher efficiency and its specifications are better than those of the Titan in almost all aspects (Table 3.1). Therefore, the two speedup values shown in each row of Table 3.2 actually represent the high and low bounds of all possible GPU-to-CPU speedups from our data. Other comparisons such as ‘Titan vs. E5’ and ‘GTX980 vs. i7’ will fall between those two values.<sup>4</sup>

In most cases, the recorded speedup beats the corresponding value reported in [1] (shown in the *Baseline* column of Table 3.2). The largest difference between the recorded speedup

---

<sup>3</sup>In all cases, the variance of the four runs is very small, indicating stable performance of both CPU and GPU code.

<sup>4</sup>Not exactly true for SMJ, but close enough as the performance of GTX980 is almost the same as Titan.

and baseline comes from the SMJ algorithm: even the smallest speedup value is a few times higher than the 2.4X reported in [1]. The NINLJ algorithm also shows a great boost of speedup over the baseline: on the higher end it reaches 20X, and even for the lower end (Titan vs. i7), everything is still higher than the 7X baseline. For INLJ, we observe speedups at about the same level as the 6.1X baseline. The HJ achieves an speedup in the range of 5.67X to 14.28X when the table size is 16M – this is much higher than the 1.9X baseline. However, there is a huge performance degradation when table size is 32M and then it goes up slowly with larger table sizes. A thorough investigation of the source code reveals the reasons for such performance drop: in the radix partitioning stage (see Section 4.1 of [1] for details), a fixed partition size is assumed. A table size bigger than 16M triggers another round of partitioning within each existing partition, resulting in a dramatic increase of total number of partitions. A prefix scan has to be done in every partition, and such scans are pure overhead for the GPU code. As the table size keeps increasing, the effects of such overhead diminish, as seen by the better GPU performance under table sizes 64M and 128M. Unfortunately, we are not able to run tests on even larger tables due to limited GPU memory. In fact, we have to stop at 64M for the GTX980. We will elaborate more on this in Section 3.4.3. Nevertheless, the above results clearly show that, other than in INLJ, the performance gap between GPU and CPU is widened in the past seven years. In other words, GPUs are more suitable for processing joins than it was in 2008.

### 3.2.2.1 Code Scalability

So far we have focused our discussions on comparing GPU with CPU. Another perspective to study the performance data is how the code scales with the growth of raw computing power of GPUs/CPUs over time. Desirably, the performance of software would *naturally* scale up with the increase of hardware capabilities in a parallel environment. To that end, we plot the relative performance (under table size 1M for NINLJ and 16M for other algorithms)

between different generations of GPUs and CPUs in Figure 3.2B, along with the relative specifications between the same set of hardware shown in Figure 3.2A. Again, the plotted GPU data represents relative performance of GTX980 to 8800 GTX, and CPU data is that of E5 to Q6600. The raw performance data of the old GPU and CPU is taken directly from [1]. In general, we can see that GPU code scales well over time - the smallest performance growth is around 4X (for SMJ). The CPU code, on the other hand, does not scale as well, especially in SMJ and HJ. For the INLJ algorithm, the CPU code scales better than the GPU code. Such results, from a different angle, explain why we achieve large GPU-to-CPU speedups in SMJ and HJ but only moderate speedups in INLJ, as reported in Table 3.2.

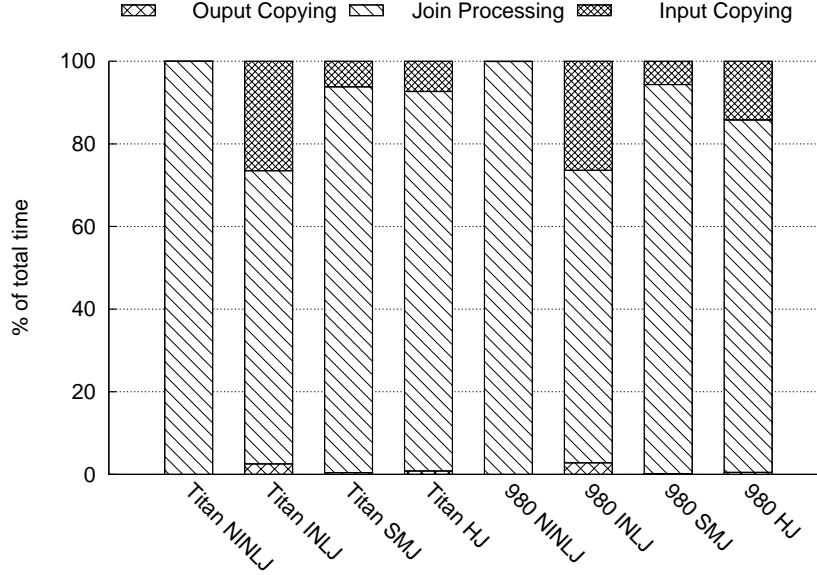
Relating the information in Figure 3.2B to the hardware information in Figure 3.2A, we also have interesting findings. All GPU algorithms scale better than the global memory bandwidth, showing the latter is not a bottleneck. Their scalability is only bound by the scale-up of compute unit capacity and L1 cache size in the GPUs – both are much larger than their CPU counterparts. On the CPU side, the scalability of SMJ and HJ performance is worse than that of all hardware resources. However, NINLJ and INLJ scale very well, indicating such algorithms are well designed.<sup>5</sup>

### 3.2.2.2 Time Breakdown

The time spent on join algorithms includes three parts: copying input from host memory, on-board join processing, and copying output back to host memory. Figure 3.3 shows the time breakdown of the tested join algorithms under two GPUs. Clearly, join processing is still the dominant component, same as shown in Figure 12 of [1]. However, the percentage of time spent on input/output data transmission between GPU and CPU is much larger in our experiments, especially in INLJ, SMJ, and HJ. In GTX 980, the numbers are 29.25%, 5.83%

---

<sup>5</sup>At this point, we are not sure why they even did better than the growth of all CPU specifications. We speculate that the compilers play a role in this – code could be much less optimized in older versions of Visual Studio based on our experience.



**Figure 3.3:** Time spent on data transmission and join processing

and 15.23%. In Titan, they are 29.06%, 6.64% and 8.17%. Both are much higher than the 13%, 4%, and 6% reported in [1]. This is caused by increased GPU performance over the years: the absolute time spent in join processing is greatly reduced (by a factor of at least 4 in Figure 3.2B). On the other hand, copying data between host and GPU is bottlenecked by the PCI-E bus, whose performance only increased by a factor of 3.

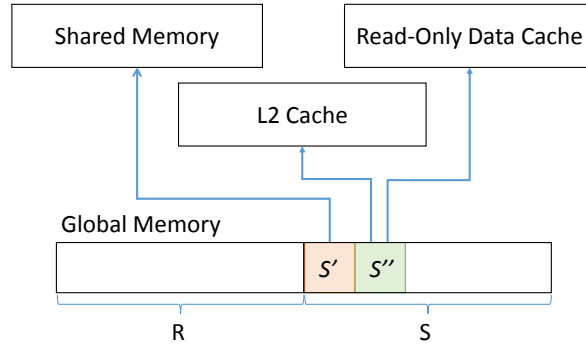
### 3.3 Optimization on New GPU Architecture

In this section, we demonstrate how features in latest GPU architectures can improve join performance. We focus on mechanisms that can be implemented without a disruptive change of the code structure. A systematic re-design of GPU join algorithms is introduced in Chapter 4.

In the Kepler architecture, the shared memory and L2 cache both come with a larger size than the 8800 GTX. Apart from that, some new features further enhance the cache system. One thing we have not mentioned in Figure 3.1 is a 48KB L1-grade read-only data cache. It is aimed at providing extra buffering for data that will not be modified during the kernel

runtime. Although the read-only cache is not fully programmable, programmers can give hints to the compiler to cache a certain piece of data in it. The Maxwell architecture [36] has no read-only cache, but the size of its L2 cache increases to 2MB.

In earlier GPU architectures, the registers are distributed to the threads running on the same multiprocessor as private storage for each thread. The contents in registers belonging to one thread could not be seen by other threads – the only way for threads to share data is via the global or shared memory. In CUDA, the basic unit of threads that are scheduled together to run on the hardware is called a *warp* – recent versions of CUDA have a fixed warp size of 32 threads. The Kepler architecture allows direct register-level data sharing among all threads in a warp by using *shuffle* instructions. A thread can disseminate its data to all others in the same warp at core speed, thus further reducing latency brought by accessing shared memory.



**Figure 3.4:** Data movement in the modified NINLJ algorithm

### 3.3.1 Cache/Register Optimization

We develop a method that increases data locality in the NINLJ program to take advantage of the L2 and read-only data cache. We present our ideas here with the help of Figure 3.4. Note that in the original NINLJ algorithm, the outer table  $S$  is divided into blocks that can fit into the shared memory. In one iteration of the outer loop, one such block  $S'$  is loaded into

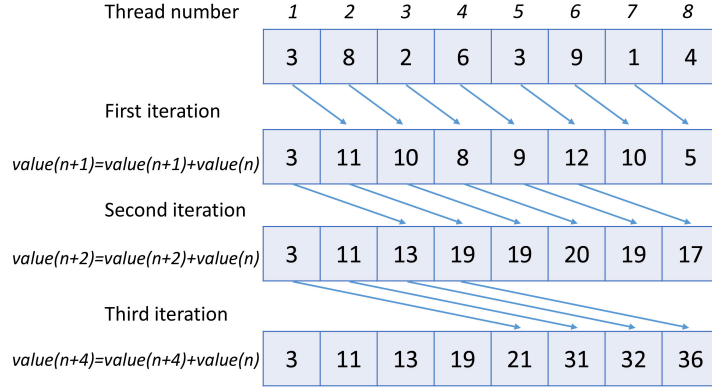
the SM and the entire inner table  $R$  is directly read from global memory. Each item in  $S'$  is accessed many times but the fact they reside in SM leads to high performance. Our strategy here is to use the read-only or L2 cache as an extension to SM by allowing another block  $S''$  to be loaded. By this, fewer rounds of reading the inner table  $R$  are needed. The challenge here is that, unlike SM, the other cache systems are not programmable. Our solution is to implement the inner loop as a nested double loop, in which loading table  $R$  is the outer layer and reading blocks  $S'$  and  $S''$  is the innermost layer. By this, we create locality such that  $S''$  will sit in the cache while seeing everything from  $R$ . There are two places for storing the extra block  $S''$ : the L2 cache and the read-only data cache. In CUDA, the later is done by putting special qualifiers before a defined pointer referencing  $S''$ .

Moreover, we reimplement the prefix-scan primitive with shuffle instructions. Note this primitive involves generating a prefix sum of numbers stored in an array (Figure 3.5), and is implemented in the original code by using shared memory. Each thread keeps an element from the array in its own register. In the  $i$ -th iteration of the kernel loop, each thread adds its element to the one that is  $i$  positions away to the right in the array. Using the CUDA `shuffle_up` instruction, such operations can be done by accessing two registers holding the two involved elements, bypassing any cache. Note that there are two limitations of the shuffle instruction: (1) registers are only open to threads in a warp; (2) it requires coordinated register access such as that in our case, random access within the warp is not allowed. After five iterations, the partial sums of each warp are collected and integrated in shared memory, which is the same as in original code.

### 3.3.2 Performance Evaluation

Table 3.3 reports the performance of L2 cache and read-only cache optimization on the GTX Titan. The two schemes achieved an average speedup of 1.3X and 1.29X, respectively. Factoring this into the GPU-to-CPU performance comparison (Table 3.2), the average Titan-





**Figure 3.5:** Data access pattern using shuffle instructions

to-i7 speedup of NINLJ now becomes 9.5X. The speedup decreases as data size becomes larger. We believe this is caused by increased cache contention – as more data is read from global memory in each iteration of the outer loop, the cached data would soon be replaced by other data. We can also see that the effects of both optimizations on performance are very similar. One might expect the utilization of both read-only and L2 cache (by putting one extra block of  $S$  into each of the two cache locations) would render even better performance. However, when we combine both techniques, the measured running time is even longer than the original code! Furthermore, the cache optimization does not yield any performance boost in GTX980. By studying the performance profiles, we found that all such results are caused by the dramatically increased number of registers assigned to each thread. As a result, the *occupancy* (i.e., number of concurrent threads running on an SMX) becomes lower, eating up the performance gain from the cache.

Table 3.4 shows the result of prefix-scan optimization by using shuffle instructions. The optimized version of prefix-scan reached a speedup of up to 1.52X over the original implementation. We notice that at 4M data size, the speedup drops to only 1.21X. This is due to underutilized computing resources since input data is too small to make full use of the computing cores and it cannot hide the kernel launch and memory access overhead. We

must point out that such boost of prefix-scan performance has a small impact on join performance - the time spent on prefix-scan is less than 1% of the total running time for most joins. However, looking forward, we believe register sharing among threads provides a novel and promising approach for code optimization in applications with coordinated data access pattern. Another fact that adds to such enthusiasm is: the size of the entire register pool in Kepler GPUs are relatively large. For example, there are 65,536 32-bit registers in each of the 15 multiprocessors of Titan. As a result, the register pool even dwarfs the L1 cache in size.

**Table 3.3:** NINLJ performance on GTX Titan under read-only and L2 cache optimizations

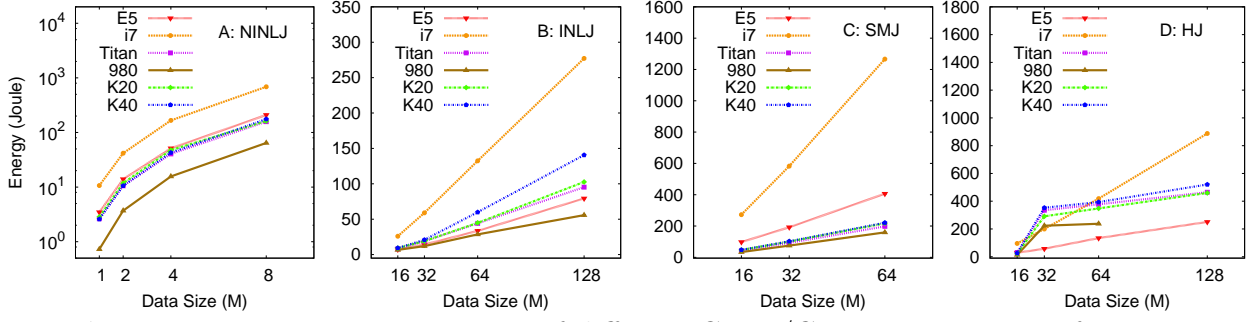
| Data<br>Size | Running time (sec) |        |           | Speedup |           |
|--------------|--------------------|--------|-----------|---------|-----------|
|              | Original           | L2     | read-only | L2      | read-only |
| 1M           | 14.64              | 11.40  | 11.62     | 1.28    | 1.26      |
| 2M           | 61.62              | 45.24  | 46.26     | 1.36    | 1.33      |
| 4M           | 252.09             | 201.71 | 197.25    | 1.25    | 1.28      |

**Table 3.4:** Running time (ms) of the prefix scan kernel optimized by shuffle instruction

| Data size | Original | With optimization | Speedup |
|-----------|----------|-------------------|---------|
| 4M        | 2.58     | 2.14              | 1.21    |
| 8M        | 4.06     | 2.67              | 1.52    |
| 16M       | 7.00     | 4.60              | 1.52    |

### 3.4 Other Considerations

In this section, we study several other related issues, in hope to provide a panoramic image of GPU’s advantages and limitations on processing joins. Specifically, we evaluate energy/power efficiency, floating point computing performance and database size. Most of the issues are mentioned in [1] but without much quantitative results.



**Figure 3.6:** Energy consumption of different CPUs/GPUs in processing four join algorithms

### 3.4.1 Energy / Power Consumption

We continuously measure the actual power consumption during the course of running the joins. Fluctuations of power are observed in all join experiments – this is due to the different hardware activities at different times of the join process. For the same exact experiments mentioned in Table 3.2, the average power consumption are shown in Table 3.5. Note that *active power* is defined as the difference between recorded system power while processing the workload and that when the system is idle. Qualitatively, we can see that GPUs consume more power than CPUs. The Xeon E5-2640, being a member of the new generation of Intel’s server-class CPU, has a much lower power profile than the older i7-3930K. On the GPU side, the GTX980 consumes less power than the GTX Titan, as energy efficiency is the main selling point of the Maxwell architecture. NINLJ consumes much more power than the other algorithms. This is due to the higher utilization of computing cores reached by this algorithm. For all algorithms, input table size does not have significant impact on power.

As to the total active energy consumption, it is obvious that in most cases the i7-3930K consumes the most energy (Figure 3.6). The GPUs are clear winners in NINLJ and SMJ algorithms, especially in SMJ where the GTX980 achieves energy efficiency one order of magnitude higher than the i7. The relatively low energy efficiency of GPUs in HJ (under large data size) is caused by their long running time rather than power consumption. Comparing

**Table 3.5:** Average active power consumption (watt)

| Algorithm | Table Size | Xeon E5 | Core i7 | GTX Titan | GTX 980 |
|-----------|------------|---------|---------|-----------|---------|
| NINLJ     | 1M         | 28.04   | 97.61   | 178.16    | 120.63  |
|           | 2M         | 28.34   | 96.01   | 179.97    | 152.51  |
|           | 4M         | 26.07   | 96.67   | 172.17    | 161.25  |
|           | 8M         | 26.75   | 100.37  | 164.83    | 165.49  |
| INLJ      | 16M        | 12.51   | 57.63   | 72.70     | 62.43   |
|           | 32M        | 11.04   | 59.57   | 78.39     | 60.95   |
|           | 64M        | 11.29   | 58.86   | 80.12     | 61.75   |
|           | 128M       | 13.37   | 55.10   | 76.84     | 52.11   |
| SMJ       | 16M        | 10.38   | 39.75   | 92.44     | 70.54   |
|           | 32M        | 10.49   | 46.69   | 95.02     | 72.38   |
|           | 64M        | 11.28   | 51.01   | 94.65     | 71.23   |
| HJ        | 16M        | 10.02   | 46.96   | 84.82     | 66.54   |
|           | 32M        | 9.97    | 48.94   | 94.37     | 67.05   |
|           | 64M        | 11.50   | 50.60   | 90.99     | 64.14   |
|           | 128M       | 10.16   | 51.75   | 86.51     | —       |

with i7, the GPUs still consume less energy in most cases of HJ. The Xeon E5 shows very good energy efficiency across the board, thanks to its low-power design. More data about energy consumption can be found in our technical report [34].

### 3.4.2 Floating Point Performance

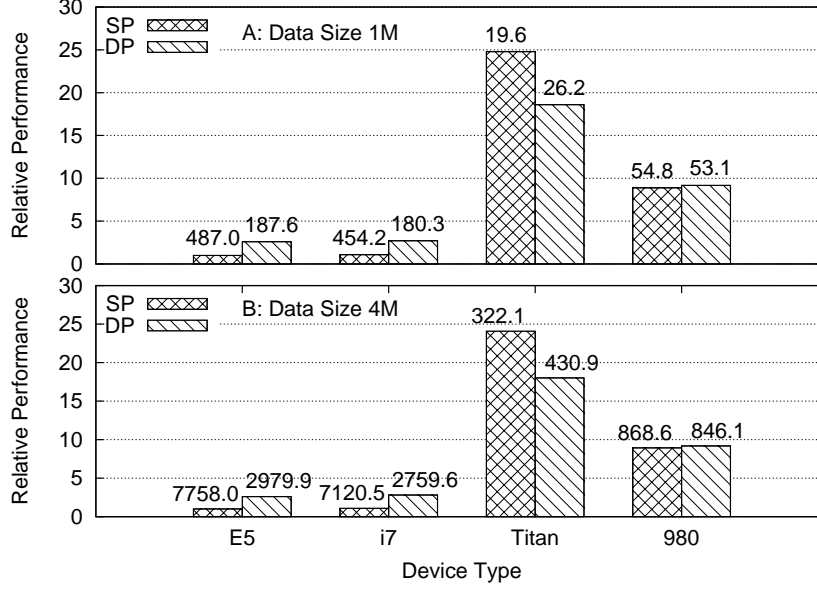
In recent few years, much progress has been made in floating point computing in GPUs. G80, the first CUDA-supported GPU, does not support floating point numbers although it has many more cores than any CPUs of its time. The following Fermi architecture supports full IEEE754-2008 single-precision (SP) and double precision (DP) floating point standards. It also features the new fused multiply-add (FMA) instructions that are much faster than the traditional multiply-add (MAD) operations. The Kepler architecture goes even further by integrating dedicated DP units into each multiprocessor [37]. This increases the peak DP performance to over 1 TFlops, roughly 1/3 of its peak SP performance. However, due

to consideration of graphics performance and power consumption, this feature is weakened in all GeForce-series gaming cards (including the GTX980) other than the GTX Titan. For example, the Titan’s DP units can operate at maximum core speed while in other Kepler cards they only run at 1/8 of the core speed.

Again, we choose the NINLJ algorithm to demonstrate the floating point performance of GPUs. Figure 3.7 shows the speedup of GPUs over CPUs by plotting the SP performance of the Xeon E5-2640v2 as the baseline (actual running time is also marked on each bar). For SP performance, the GTX Titan achieves a surprising 24X and 23X speedup over the Xeon E5 and Core i7, respectively. This result doubles its speedup over the CPUs with integer key values (Table 3.2). For DP performance, Titan reaches about 7X speedup over both of the CPUs, which is roughly the same as integer-based results reported in Table 3.2. The main reason for such different speedup is that both CPUs performed much better in DP than in SP computing. Their SP performance is only 1/4 of their integer performance while their DP performance is around 1/2. Meanwhile, the performance of GTX Titan only degrades by half for both SP and DP. This reflects the different strategies adopted in CPU and GPU hardware design – much more resources are dedicated to DP computing in CPUs. The GTX980 is less powerful in floating point computation, yet it still achieves a 8-9X speedup in SP and a 2-3X speedup in DP over the CPUs.

### 3.4.3 Limitation of Memory Size

The GPU join algorithms we tested assume all input /output data and intermediate results can be stored in the global memory therefore the size of the latter determines how large the input tables can be. To explore the space use of GPU joins, we repeatedly run the code with a varying table size (in a binary search manner) until we find the largest table each algorithm can run with. The largest table we can run each join in the GTX Titan (with 6GB of global memory) is as follows: with a larger state (i.e., both sorted tables) to keep, the SMJ



**Figure 3.7:** Relative performance of NINLJ with SP/DP keys in different CPUs/GPUs under two table sizes

will stop at 96 million records in both tables (i.e., 1.5GB total data size). Following that are HJ and INLJ – the largest table they can run have 200M and 250M records, respectively. This makes sense as the HJ and INLJ only keep intermediate state with a size equivalent to one of the input tables. We did not obtain data for NINLJ as each run of it needs excessively large amount of time. We believe the allowed table size will be larger than that of INLJ (we tried 256M records without a problem) as there is almost no intermediate data other than the output table. With only 4GB of global memory, smaller tables are allowed in the GTX980. However, the order of reachable table size does not change for the algorithms: SMJ, HJ, and INLJ have maximum table sizes of 64M, 121M, and 185M, respectively.

### 3.5 Discussions

In this section, we summarize our findings and comment on the advantages and limitations of GPU-based join processing. In particular, our discussions will directly respond to the issues raised by He *et al.* in Section 6 of [1].

### 3.5.1 Main Findings and Recommendations

The hardware resources on GPUs have expanded rapidly over the past few years. This provides increasingly stronger support of data-parallel join processing and builds the foundation of much higher performance than those reported in 2008. We also notice that the capacity growth of GPUs is unbalanced between its compute cores and global memory bandwidth (i.e., 13X vs. 3X as shown in Figure 3.2A). Such a strategy in GPU design, although suitable for high-performance computing (HPC) applications, leaves a question mark on whether join processing can really make good use of GPGPU. Generally, the performance bottleneck of database operations such as join and selection is memory access given that the latency of memory system is hundreds of CPU clock cycles [38] and the demand on arithmetic operations is small by the nature of such operations. GPUs share the same problem although its GDDR5 global memory system has higher bandwidth and lower latency than the DDR3 host memory. Therefore, the GPU-to-CPU speedup is not expected to exceed 8X, which is roughly the difference between the memory bandwidth of today’s mainstream GPUs and CPUs (Table 3.1). To our surprise, the performance of NINLJ, SMJ, and HJ (considering only 16M input) is way better than that on the GTX980. The key to such success is clearly the large cache size, which effectively moved the bottleneck away from global memory. In an extreme case of NINLJ, global memory utilization dropped to less than 1% and arithmetic unit utilization reaches up to 84%! We are pleased to see that increasing memory bandwidth and size (by three orders of magnitude) is the main design goal of Pascal - Nvidia’s next generation GPU architecture [39].

As to program development, it is still true that GPU code has to be written from scratch due to the different programming models between CPUs and GPUs. As more and more programmers are trained in GPGPU programming, this does not seem to be as big a concern as before. We believe the rapid change of architectural design is a major inconvenience in

CUDA programming. New features emerge in each new generation of GPU architecture. Our results show that an algorithm designed for older GPUs may not fully utilize resources in newer ones. It is important to (at least partially) re-design the algorithm considering the new architectural features. There are also problems in compiler support of new features. For example, the same shuffle instruction code that work perfectly in Titan (Section 3.3) cannot be compiled when the GTX980 is chosen as the target device. However, we must emphasize that new GPU features can bring great performance benefits.

### 3.5.2 Response to Concerns in [1]

Algorithm design and optimization in GPGPU is still a complex task. In particular, the random data access pattern of the SMJ, INLJ, and HJ algorithms poses a threat to GPU join performance. The SIMD architecture makes a GPU vulnerable to high latency caused by code divergence. We however want to point out that in CUDA, the direct impact of divergence is within a single warp. With higher level of parallelism made possible by the abundant resources in modern GPUs, memory stall can be effectively hidden. Recall that the SMJ and HJ algorithms both perform well on the new GPUs. Atomic operations are now supported in CUDA, it can effectively handle read/write conflicts. That said, the pre-scan routines to determine write offset in the join algorithms cannot be replaced by atomic operations, as dynamic memory allocation is not allowed in current version of CUDA.

High power efficiency has been a major goal of GPU design, as is in CPUs. We have witnessed a sharp drop of power consumption in the recent two generations of Nvidia GPUs. We admit modern CPUs (e.g., the E5-2640 we used) have become extremely power efficient, and there is still room for improvement for GPUs. However, by putting performance into the equation, we see that GPUs are obvious winners in energy efficiency (Section 3.4.1). Our experiments (e.g., comparing i7-3930K with E5-2640) imply that high power efficiency comes with the cost of a large performance cut in CPU design.



Finally, the situation of limited data type support has changed a lot. Floating-point numbers are not only supported by the CUDA language, the new GPU hardware also dedicates much of its silicon to speed up floating-point computation. This is a natural result of the GPU industry's vision to make GPGPU the core technology in HPC systems. Our work shows that performance of joins with SP and DP keys is many times higher than the CPUs (Section 3.4.2).

## Chapter 4: Fast In-Core Join Algorithms on GPUs: Design and Implementation

<sup>6</sup> The multitude of modern parallel computing platforms has provided opportunities for data management systems and applications. While CPUs are still the most popular platform for implementing database management systems (DBMSs), GPUs have gained a lot of momentum in doing the same due to its computing power, high level of parallelization, and affordability. In this chapter, we present our recent work in the context of a GPU-based data management named G-SDMS [24]. In particular, we focus on the design and implementation of relational join algorithms. Our goal is to develop GPU-based join code that significantly outperform those found in literature [1, 11, 3, 4, 5, 40, 14, 22].

In the past few years, in addition to the computing capacity that has grown exponentially, the GPUs have undergone a dramatic evolution in hardware architecture and software environment. On the other hand, existing join algorithms are designed for earlier GPU architectures therefore it is not clear whether they can make the most out of latest devices in the market. Although the GPU code may scale well with the increasing amount of computing resources in newer GPU devices, maximum performance cannot be achieved without optimization towards new GPU components and features in the runtime system software. Our analysis and empirical evaluation of existing GPU join algorithms confirmed such reasoning [26]. Therefore, the objective of our work reported in this chapter is a novel design of join algorithms with high performance under today’s GPGPU environment. In particular, we

---

<sup>6</sup>This chapter was published in Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM ’17). Association for Computing Machinery, New York, NY, USA, Article 17, 1–12. doi: 10.1145/3085504.3085521. Permission is included in Appendix A

overhaul the popular radix hash join and redesign sort-merge join algorithms on GPUs by applying a series of novel techniques to utilize the hardware capacity of latest Nvidia GPU architecture and new features of the CUDA programming framework. As a result, while our implementation borrows code for common data primitives (e.g., sorting, searching and prefix scan) from popular CUDA libraries, our algorithms are fundamentally different from existing work.

Our hash join is based on the well-known radix hash join. We used a two-pass radix partitioning strategy to reorganize the input relations. In order to increase hardware utilization, we keep a shared histogram in the shared memory for each thread block and all threads in the same block update the shared histogram via atomic operations. This reduces the usage of shared memory per thread therefore allows for more concurrent threads working together. We also assign multiple works per thread by loading more data into the large register file in the new GPU architecture. By doing this each individual thread improves instruction-level parallelism and higher overall efficiency is achieved. Previous work [1, 15] requires two scans of the inputs before writing the output to memory. To remove this large overhead, we propose an output buffer manager that enables probe in only one pass. With the help of efficient atomic operations, threads acquire the next available slot from the global buffer pointer and output independently. Finally, we take advantage of the convenient Dynamic Parallelism supported by the latest CUDA SDK to dynamically invoke additional threads to tackle skewed partitions without additional synchronization and scheduling efforts.

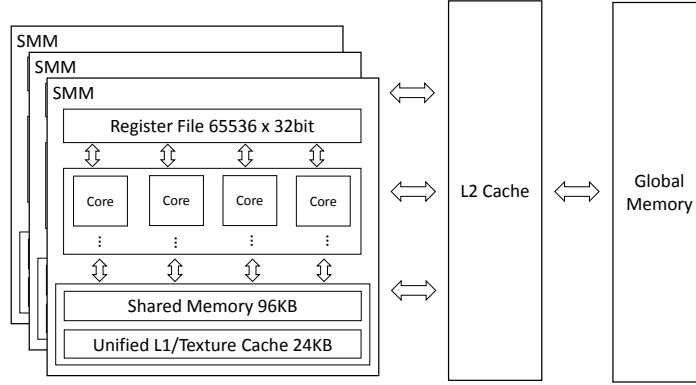
Our sort-merge join algorithm shares the same idea of using registers to allow more work per thread. Apart from that, our implementation heavily relies on an efficient parallel merge algorithm named Merge Path [41, 42] in both sort and merge stages. Merge Path partitions the data in such a way that threads can work independently with balanced load. With a linear total work efficiency, Merge Path is faster than traditional parallel merge algorithm that requires a binary search for each tuple. The sort algorithm is designed in a hierarchical

manner. First, each thread sequentially sorts their own chunk of data in register. Then all the threads in the same block work together to merge their data into a list staying in shared memory. After that, all the thread blocks combine their data in the same manner in global memory. It is obvious that this method makes full use of the memory hierarchy of the GPU, especially the register file and shared memory.

We also extend our designs to the scenario of large tables that cannot fit into the GPU global memory. This is an aspect that is largely unexplored in existing work. Our strategy is to maximize the overlap between the transmission of partitions of input tables and the processing of resident data. By using CUDA streams, we divide the single workflow into two pipelines so that input data transfer and kernel execution can overlap.

Experiments show that our new hash join obtains a 2.0X to 14.6X speedup over the best implementation known to date, while the new sort-merge join achieves a speedup of 4.0X to 4.9X. Statistics provided by CUDA Visual Profiler also show that our new algorithms achieve much higher multiprocessor occupancy, higher shared memory bandwidth utilization and better cache locality. Compared with the latest CPU code, our hash join and sort-merge join are respectively up to 5.5X and 10.5X as fast. When handling data larger than the GPU device memory size, our new algorithms achieves 3.6-4.3X and 11-12.8X speedup in hash join and sort-merge join, respectively.

This chapter makes the following contributions. First, we design and implement GPU-based join algorithms by optimizing various stages of sort merge and hash joins on the latest GPU architecture. Comparing with previous GPU join algorithms, our code achieves a large speedup, and the utilization of GPU resources increases considerably. It is safe to say that our join code represents the current state-of-the-art in this field. Second, we present a design of GPU joins that reduces I/O overhead in dealing with input tables that cannot be stored in GPU memory. To the best of our knowledge, this is the first reported work in joining tables beyond the memory size of GPU devices. Finally, we carry out a thorough comparison of



**Figure 4.1:** Layout of latest NVidia GPU architecture

the performance of GPU-based join algorithms and their CPU counterparts. In addition to the conclusion that GPU-based algorithms are superior over best known CPU counterparts, we provide an anatomy of such algorithms to interpret the observed results.

## 4.1 Join Algorithm Design on GPUs

In this section, we introduce the recent development of GPU architecture, and then highlight hardware and software features that are most relevant to join processing. Based on that, we present new GPU hash and sort-merge join algorithms that take advantage of such features to effectively utilize GPU resources.

### 4.1.1 GPU Architecture

Before we discuss GPU joins, it is necessary to sketch the main components of the GPGPU environment we work on. In this chapter, we focus on NVidia GPU devices and the CUDA programming model. The layout of the latest NVidia GPU (e.g., Maxwell and Pascal) architecture is shown in Figure 4.1. Such a GPU consists of a few multiprocessors, each of which contains 128 computing cores, a large register file, shared memory and cache system. In CUDA, the threads are grouped into thread *blocks*. Each block runs on one multiprocessor,

and 32 threads form a basic scheduling unit called a *warp*. A block may contain several warps. The threads are scheduled in SIMD manner where a warp of threads always execute the same instruction but on different data at the same time.

The memory hierarchy in the GPU also has different scopes. The variables of a thread are stored in the register file and private to that thread. However, CUDA provides *shuffle* instructions that allow threads in the same warp to shared data in the registers. At block level, *shared memory* is a programmable L1-level cache that can be used for fast data sharing among threads in the same block. The *global memory*, or device memory, serves as the main memory for GPU. Although it provides up to a few hundreds GB/s of bandwidth, coalesced memory access is needed to fully utilize the bandwidth. There is also an L2 cache that buffers the global memory access for the multiprocessors.

#### 4.1.1.1 *New Features of GPUs*

The hardware design of GPUs has experienced drastic changes in recent years. This has deep impacts on our join algorithm design and implementation.

First, the number of computing cores increases steadily, giving rise to much higher GFLOPs of the GPU. The Titan X has nearly 30X more cores than that in 8800GTX, but CPU core counts only increase by 4-5X during the same period of time. Apart from the quantity, the organization of the multiprocessor has also changed over time. For example, one multiprocessor in Maxwell consists of 128 computing cores divided into four blocks. Each block of cores has dedicated scheduler with dual issue capability. This improves the efficiency of scheduling, power consumption and chip area, but requires more parallelism to achieve high utilization.

An important change is the large number of registers starting from Kepler architecture. Each multiprocessor has 64K 32-bit registers, resulting in 256KB capacity, which is larger than that of L1-level cache! This implies that the register file can hold larger amount of

data, hence more work per thread is made possible at register speed. Data in registers had been set to be private to each thread, but now they can be shared among threads within the same warp via *shuffle* instructions.

Atomic operations are widely used in parallel algorithms to operate on shared data or to gather results. In early GPUs, atomic operations are supported via a locking mechanism. It is improved in Kepler via native atomic operations in global memory, and the affected memory addresses are aggressively cached (in L2 cache). Maxwell and Pascal go one step further by supporting them in shared memory. This improvement simplifies applications that need to update shared counters or pointers, and more importantly, relieves a major performance bottleneck associated with atomic operations due to the high bandwidth of shared memory.

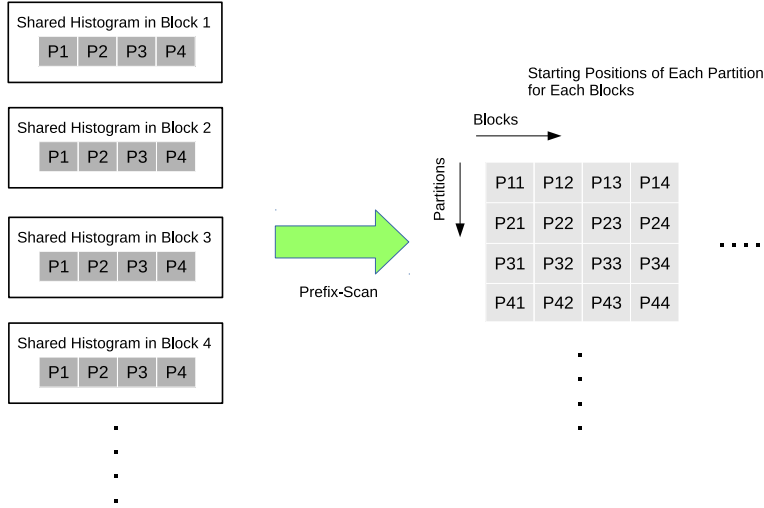
Dynamic parallelism is another new feature available starting from Kepler. It allows an active kernel to launch other kernel calls, thus dynamically creating additional workload when the parent kernel is running. This feature enables recursive kernel calls which is not possible in earlier generations of GPUs. We will discuss in detail on how we use this feature to tackle the data skewness problem in hash join.

Creating overlaps between the processing of *in situ* data and shipping of new data inputs/outputs is a key technique in joining large tables. Such concurrency of different activities are made possible by a CUDA mechanism called *CUDA stream*. In presenting our algorithm design, we first assume the input tables can be completely placed in global memory, then we remove that assumption in Section 4.1.4.

#### 4.1.2 Hash Join

Our hash join is based on the popular idea of radix hash. The process consists of three parts: partitioning input data, building hash table and probing. However, we adopt the idea used in [1] that by reordering the tuples in a relation according to its hash value, the

partitioning and building stages are combined into one. Therefore, the tuples with the same hash value are clustered into a continuous memory space, which ensures coalesced memory access when threads load data from a certain partition. Despite this, our hash join algorithm implementation is fundamentally different from the design reported in [1] in most parts.



**Figure 4.2:** Shared histogram used in Partitioning and Reordering in GPU hash join

#### 4.1.2.1 The Partitioning Stage

The partitioning stage starts with building histograms for hash values to reorder the tuples of both input tables. In previous work, a thread reads and processes one tuple at a time because the multiprocessor has very few registers. This method is straightforward but is less capable of hiding latency via instruction-level parallelism. To utilize the large register file in new GPU architecture, our implementation loads VT (short for *values per thread*) tuples into registers of the thread all at once so that each threads are assigned more workload at the beginning. This increases the instruction-level parallelism within each thread, and the memory access can be overlapped with computation to hide latency. Each thread processes its own data independently and updates the shared histogram in shared memory (Fig. 4.2).



Being different from the method in [1], where each thread keeps private histograms for each partition in shared memory, our algorithm keeps only one shared copy of histogram in each thread block, as Algorithm 1 shows<sup>7</sup>. In early generation of GPUs, atomic operations are either not supported or involve considerable overhead. It was not feasible to update shared histogram among a number of threads. The problem with keeping private histograms in each thread is that it would consume too much shared memory when either the number of threads in each block or the number of partitions is high, reducing the number of active threads running on each multiprocessor (i.e., called *occupancy*). This might not be a serious issue in old devices such as 8800GTX. Since they only have 8 cores per multiprocessor, a small number of threads are enough to keep it busy. However in newer architectures, more concurrent threads are required to keep the hardware at optimal performance. By using one shared copy of the histogram, the amount of shared memory consumed by a block is reduced by a factor that equals the block size, and is no longer depending on the number of threads in a block, resulting in more active threads for multiprocessors. Also thanks to native atomic operation support on shared memory in Maxwell and Pascal, all the threads in a block can update the shared histograms with a very small overhead.

---

**Algorithm 1:** Histogram in GPU Hash Join

---

**Require:** Relation  $R$

**Ensure:** array of histograms  $SharedHisto[]$

- 1: Initialize  $SharedHisto[nPartitions]$  to 0;
  - 2:  $data[VT] \leftarrow$  load VT tuples from relation  $R$ ;
  - 3: **for**  $i = 0$  to  $VT-1$  **do**
  - 4:    $h \leftarrow Hash(data[i].key)$ ;
  - 5:    $atomicAdd(SharedHisto[h], 1)$ ;
  - 6: **end for**
  - 7: Write  $SharedHisto[nPartitions]$  to global memory;
- 

<sup>7</sup>All pseudocode is written from the perspective of a single thread, following the Single-Program-Multi-Data (SPMD) programming style in CUDA.

In previous work, a multi-pass radix, or a variable number of pass partition is used. However, in this method we found there is a non-linear growth of number of partitions with the table size increasing. This results in a non-linear execution time increase. We adopt a two-pass radix partition mechanism in our implementation. We keep the partition size to be small enough (e.g., less than 512 tuples for each thread block) to fit into shared memory, therefore the probe stage only needs to read the data once from the global memory. To achieve such small partition for large input, we have to create a large number of partitions. If a single-pass method is used, the shared memory is not able to hold that many histograms. Thus, we use a two-pass method where the first pass reorganizes the input into no more than 1024 partitions, and the second pass further divides the partitions from the first pass into smaller ones. By using this method, we can process a single table containing 500 million pairs of integers (key+ value). This is a reasonable size since in our experiment the Titan X with 12GB memory can hold two 128 million-tuple arrays plus intermediate data.

To reorder the tuples (Algorithm 2), each thread block has to know its starting positions of the partitions. The shared histograms are copied to global memory. Then a prefix scan is performed to determine the starting position of all the partitions for each block (Fig. 4.2). Once the positions are obtained, all the threads can reorder the tuples in parallel by atomically incrementing the pointers for each partition. Since our method uses shared histogram and its prefix sum, the writing positions of the threads in the same block are also localized. This increases locality of memory access, thus the cache would be in use to buffer the writes.

#### 4.1.2.2 The Probe Stage

In the *probe* stage (Figure 4.3), each partition of input table  $R$  is loaded into shared memory by one block of threads. A partition of the other table  $S$  with the same hash value is loaded into registers by the same threads. This is the same mechanism mentioned

---

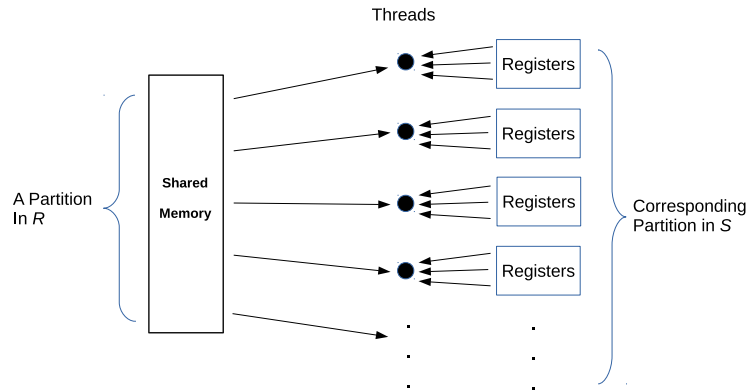
**Algorithm 2:** Reorder in GPU Hash Join

---

**Input:** relation  $R$ **Output:** reordered relation  $R'$ 

- 1: SharedHisto[nPartitions]  $\leftarrow$  load the exclusive prefix sum of the histogram from global memory;
  - 2: Synchronize;
  - 3: data[VT]  $\leftarrow$  load VT tuples from relation  $R$ ;
  - 4: **for**  $i = 0$  to VT-1 **do**
  - 5:    $h \leftarrow \text{Hash}(\text{data}[i].\text{key})$ ;
  - 6:   //get current writing position and then increment
  - 7:    $\text{pos} \leftarrow \text{atomicAdd}(\text{SharedHisto}[h], 1)$ ;
  - 8:    $R'[\text{pos}] \leftarrow \text{data}[i]$ ;
  - 9: **end for**
- 

in previous section, thus every access to partitions of  $S$  is at register speed. To write the outputs back to memory, the traditional wisdom (as in [1] and even CPU work such as [15]) is to perform the probe twice. The first probe returns the number of outputs for each partition to determine the location of the output buffer for writing outputs. The total number of outputs and starting position of each partition is obtained by a prefix scan of these numbers. Given the number of outputs, the output array can be allocated and then the second probe is performed to actually write the output tuples. This scheme eliminates the overhead of synchronization and dynamic allocation of buffers, and efficiently outputs in parallel by doing more work. The pseudocode of such a design of probe is shown in Algorithm 3.



**Figure 4.3:** Workflow of threads of probe stage in hash join

---

**Algorithm 3:** Probe in GPU Hash Join

---

**Input:** relations  $R$  and  $S$

**Output:** array of matching pairs  $globalPtr$ ; number of matches for each block  $matches$ ;

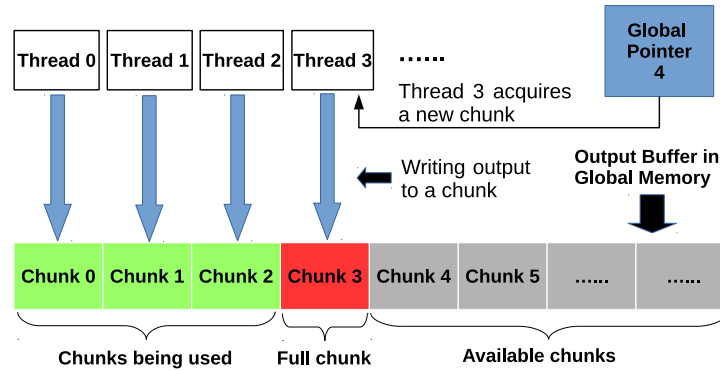
```
1: pid  $\leftarrow$  blockIdx.x; //Partition id
2: while pid < nPartitions do
3:   matches  $\leftarrow$  0;
4:   SharedBuf[VB]  $\leftarrow$  load partition pid of  $R$ ;
5:   Synchronize;
6:   data[VT]  $\leftarrow$  load VT tuples from partition pid of relation  $S$ ;
7:   bufPtr  $\leftarrow$  atomicAdd(globalPtr,bufSize);
8:   count  $\leftarrow$  0;
9:   for i = 0 to VT-1 do
10:    for j = 0 to VB-1 do
11:      if Hash(data[i].key) == Hash(SharedBuf[j].key) then
12:        bufPtr[count++]  $\leftarrow$  (data[i],SharedBuf[j]);
13:        if count == bufSize then
14:          bufPtr  $\leftarrow$  atomicAdd(globalPtr,bufSize);
15:          count  $\leftarrow$  0;
16:        end if
17:      end if
18:    end for
19:  end for
20:  pid  $\leftarrow$  pid + NumBlocks;
21: end while
```

---

However, we realize that the overhead of probing twice is too high. To reduce such overhead, we design a *buffer management* mechanism in which threads directly output to different locations of a buffer pool in global memory (Fig. 4.4). We first allocate an output buffer pool of size  $B$  and divide it into small pages of size  $b$ . A global pointer  $P$  holds the position of the first available page in the buffer pool. Each thread starts with one page and fills the page with output tuples by keeping its own pointer to empty space in the page. Once the page is filled, the thread acquires a new page pointed to by  $P$  via an atomic operation and increment  $P$ . With the direct output buffer, threads can output directly in the probe stage in parallel and no complex synchronization is needed. We basically trade the cost of acquiring new pages for elimination of the second probe. Since the atomic operation only happens when a page is filled, we expect little conflicts in accessing the global pointer  $P$ .

Plus, we can adjust the page size  $b$  to reach the desirable tradeoff between such overhead and buffer space utilization (i.e., larger page reduces overhead but may render more empty space within pages).

To tune the output buffer even more aggressively, an alternative is to divide the whole output buffer into chunks. Each thread block is assigned one chunk for output their results. Each block keeps a pointer in the shared memory that redirects to the next available slot in the output chunk. When a thread in a block needs to output, it acquires the current value of the pointer in the shared memory and increases it via an atomic operation, then it outputs the result to the available slot. This technique will take advantage of low cost of atomic operations against shared memory locations.



**Figure 4.4:** A case of direct output buffer for GPU hash join, showing Thread 3 acquiring chunk 4 as output buffer

#### 4.1.2.3 Skew Handling

Our hash join design takes data skew into consideration. Here by “data skew” we mean some of the partitions based on the hash value can be larger than others. In extreme cases, most of the data are distributed in just a few partitions. As a result, the corresponding thread blocks in the probe stage become the bottleneck of the whole procedure. To deal with data skew, previous work processes these skewed partitions in a separate kernel function that

provides more working threads for the extra data. This method is simple and efficient, but needs to keep more intermediate states for scheduling.

In our implementation, we take advantage of dynamic parallelism that was introduced since Kepler architecture. This technique allows dynamic creation of additional kernels within current workflow. If the size of a certain partition exceeds the predefined threshold, the block that is processing this partition creates a child kernel that exclusively works on this partition. The child kernel runs concurrently with the parent kernel and other child kernels until it finishes. Then it returns to its parent thread. We can dynamically change the launching parameters of the child kernels (i.e. block size and grid size) according to the sizes of their corresponding partitions. This technique brings more flexibility for dealing with skewed data.

#### 4.1.3 Sort-Merge Join

As usual, sort-merge join is divided into two stages: (1) sorting the input relations by the attribute(s) involved in the join condition; and (2) merging the two sorted relations to find matching tuples.

##### 4.1.3.1 The Sort Stage

Our program features a highly efficient parallel merge-sort algorithm. Previous work often implements radix sort [43] or bitonic sort [1] that are also suitable for parallel computing. However, they both have limitations in that the radix sort only applies to numeric data and it becomes costly as the key size grows, while the bitonic sort has a unique pattern of comparison which requires power-of-two number of data points. Merge-sort can sort any type of data and are more flexible on data size than bitonic sort. Although bitonic sort in serial code has low time complexity ( $O(\log^2 n)$ ), its best parallel version has a subpar  $O(n \log^2 n)$  total computation [44]. It is also hard to exploit locality and coalesced memory access when



**Figure 4.5:** Parallel merge with 7 threads using Merge Path

data is large as it accesses different locations each time. Merge-sort, on the contrary, merges two consecutive chunk of data at a time, which can utilize the register blocking, coalesced global memory access and shared memory of the GPU. According to our experiments, this highly efficient use of memory bandwidth results in a 7X speedup compared with the bitonic sort in existing work.

Our sort is based on a parallel merge algorithm named Merge Path [41, 42], the main idea of which is shown in Fig. 4.5. Consider the merge of two sorted arrays A and B, a merge path is the history of the merge decisions. It is more clearly illustrated by a  $|A| \times |B|$  matrix, in which an element  $(i,j)$  is 1 when  $A[i] < B[j]$ , and 0 otherwise. We can obviously see that the merge path lies exactly on the boundary between the two regions containing only 0s and 1s, respectively. If we break the merge path into equal-sized sections, the projections of each section on A and B arrays correspond to the elements to be merged by this section, thus each section can merge their own data independently. The most essential part in this method is how to find the merge path without actually carry out the merging process. To find the merge path, we need the help of cross-diagonals, which are the dash lines in Fig. 4.5. By performing binary searches on the pairs of  $A[i]$  and  $B[j]$  along the cross-diagonals of

the matrix, where  $i + j$  equals to the length of the corresponding cross-diagonal, we obtain the intersections of the merge path and the cross-diagonals. These intersections provide the starting and ending points of each sections of the merge path. As the sections are equal-sized, load balancing would be naturally achieved without additional effort. Based on this highly parallel and load-balanced merge procedure, efficient merge-sort algorithm can be realized on GPUs.

In our sort stage, input relations are first partitioned into small chunks of size VT. Then each thread loads a chunk of input data into its registers as an array using static indexing and loop unrolling to achieve more efficiency, as shown in Algorithm 4. That is to access the array using for loops in a sequential way. This method ensures the whole chunk resides in registers as long as the number of registers needed does not exceed 256 per thread. Each thread performs sequential odd-even sort on its own chunk and stores the sorted chunks into shared memory. Since VT is set to 8 after some tests for optimal performance for the GTX Titan X, the overhead of using odd-even sort on data sitting in registers is acceptable. After each thread has their own chunk sorted, all the threads in a thread block work cooperatively to merge the chunks in shared memory using Merge Path until they become a single sorted array. Then all the blocks store their outputs to global memory and cooperatively merge the arrays using Merge Path again, until the whole relation is sorted (Algorithm 5). The arrays are loaded into the shared memory, and each thread executes serial merge independently on their own partitions, and stores the merged list to registers which is to be output later to global memory in batch. In summary, our sort stage relies heavily on registers (in BlockSort) and shared memory, which were of much smaller volume in early GPUs.

#### 4.1.3.2 The Merge Join Stage

In the merge join stage, the two sorted relations are treated as if they were to be merged into one list. Previous work first partitions relation  $R$  into small chunks that fit into the



---

**Algorithm 4:** BlockSort

---

**Input:** Input relation  $R$ ;

**Output:** Sorted sublists;

1:  $\text{data}[\text{VT}] \leftarrow \text{load VT tuples from relation } R$ ;

2: sort  $\text{data}[\ ]$  sequentially;

3: copy  $\text{data}[\ ]$  to shared memory;

4: **for**  $n \leftarrow 2, 4, 8, \dots, \text{BlockSize}$  **do**

5:    $L \leftarrow \text{VT} \times n / 2$

6:   find the merge path of two sorted  $\text{data}[\ ]$  of length  $L$ ;

7:   merge the two sorted  $\text{data}[\ ]$  into one list of length  $2L$  in shared memory with  $n$  threads cooperatively;

8: **end for**

9: Store the sorted tuples to global memory;

---

shared memory, then searches the other relation  $S$  for matching chunks. Each tuple in a chunk of  $S$  finds matches using binary search on the corresponding chunk of  $R$ .

In our implementation, the Merge Path method is used at this stage as well. To find matching tuples, we start from partitioning the input relations using merge path so that each thread can work on individual chunks of the input. After loading the corresponding chunks from the two inputs into register, each thread loops over each elements of  $R$  and runs merge path to find the starting point (e.g. the lower bound) of matching in  $S$ . This procedure resembles a serial merge of two sorted lists, thus the total work of all threads is linear to the number of inputs. The second step is similar to the first one, except that this step is to find the starting point of matching of  $R$  for each elements in  $S$ , which is exactly the ending point (e.g., the upper bound) of matching in  $S$  for tuples in  $R$ . By subtracting the starting position from the ending position, the number of matches for each tuple in  $R$  is obtained. Before output results, a prefix scan on the array of number of matches gives the total size for allocating output buffer. Since we know where to find the matches, a second scan is no longer needed in the output stage.

---

**Algorithm 5:** Merge Data from different blocks

---

**Input:** sorted sub-arrays of size  $VT \times \text{BlockSize}$ ;  
**Output:** a single sorted list;  
1:  $VB \leftarrow VT \times \text{BlockSize}$ ;  
2: **for**  $n = 2, 4, 8, \dots, \text{NumBlocks}$  **do**  
3:    $L \leftarrow VB \times n / 2$ ;  
4:   find the merge path of two sorted sub-arrays of length  $L$ ;  
5:    $\text{dataShared}[VB] \leftarrow$  corresponding partitions of  
    sub-arrays for current block;  
6:   merge the tuples in  $\text{dataShared}[\ ]$  into one list of length  
     $2L$  to registers;  
7:   store the sorted list to global memory;  
8: **end for**

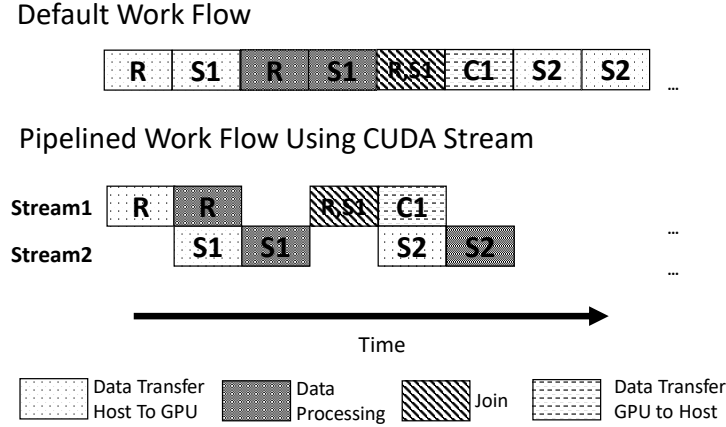
---

#### 4.1.4 Handling Large Input Tables

So far we have made the assumption that both tables as well as the intermediate results of the join can be put into the GPU global memory. This sets a limit on the size of tables that can be processed. In this section, we report our efforts in removing that assumption. Following the ideas of disk-based joins, we can obviously break the input tables into chunks and process pairs of chunks (one from each table) in a GPU using the aforementioned join algorithms. Join results of each pair of chunks are written back to host memory.

The first aspect is how to schedule the shipping / processing of different data chunks to/in the GPU. Note that a thorough study has to consider the relative table sizes and the number of GPU devices. In this chapter, we focus on the following scenario: there is only one GPU, table  $R$  can be completely stored in the global memory while table  $S$  is of an arbitrarily (large) size. Such a scenario represents a typical business database design such as the one found in TPC-H. Furthermore, solutions developed for such will build the foundation for more complex scenarios. Given that, we first load  $R$  entirely into GPU, and join  $R$  with each and every chunk of  $S$ , and ship results back to host memory. Apparently, as  $R$  resides in GPU, we conduct the first stage (e.g., partition, sorting) of the join only once for  $R$ .

Another aspect is to hide the data shipping latency with join computation on the device. In particular, we take advantage of the CUDA Stream mechanism to allow concurrent data transfer and kernel execution between neighboring rounds of chunked joins (Fig. 4.6). Specifically, each chunked join involves a kernel launch, and the series of kernel launches are encapsulated into CUDA streams. After table  $R$  is transmitted to GPU memory, the kernel for processing (i.e., sorting or building hash)  $R$  and the transfer of  $S_1$  are issued simultaneously. When the join results  $C_1$  are being written back to the host, the shipping of  $S_2$  happens at the same time. In this way, the work flow is pipelined and the overlapping of kernel execution and data transfer helps reduce the total running time.



**Figure 4.6:** Overlapping data transmission and join processing using two CUDA streams

We also worked on the scenario of processing joins in multiple GPU devices. It involves innovative data transmission scheduling among the different GPU cards as well as between the card and host. Note that the two types of transmission are done in different physical PCI-E channels therefore we can handle cases in which one table can only be placed in multiple GPUs without much performance penalty. Due to page limits and the complex techniques involved, we unfortunately have to skip such details. We leave the study of joins between very large tables (such that neither table is smaller than the aggregated memory size of multiple GPUs) as future work.

**Table 4.1:** Specifications of hardware mentioned in this chapter

| Device  | CPU            |              | GPU                     |                         |
|---|----------------|--------------|-------------------------|-------------------------|
|   | Xeon E5-2630v3 | Xeon E5-2670 | Maxwell Titan X         | Kepler Titan            |
| Clock Rate  | 2.40GHz        | 2.60GHz      | 1.00GHz                 | 0.84GHz                 |
| Core counts   | 8              | 8            | $24 \times 128$         | $14 \times 192$         |
| L1 Cache  | 256KB          | 256KB        | $96\text{KB} \times 24$ | $64\text{KB} \times 14$ |
| L2 Cache  | 2MB            | 2MB          | 3MB                     | 1.5MB                   |
| L3 Cache  | 20MB           | 20MB         | –                       | –                       |
| Memory*   | 128GB DDR4     | 64GB DDR3    | 12GB GDDR5              | 6GB GDDR5               |
| Memory Bandwidth *  | 59GB/s         | 51.2GB/s     | 337GB/s                 | 288GB/s                 |
| Max GFLOPS  | 153.6          | 166.4        | 6144                    | 4494                    |
| * For CPUs, here we refer to the host memory of the computer.<br>For GPUs, we mean the global memory. |                |              |                         |                         |

## 4.2 Evaluations

We evaluate the performance of our GPU-based join algorithms by comparing them with existing GPU and latest CPU join code. In addition, we also show the effects of different factors on the performance. The hardware and software configurations are described in Section 4.2.1.

### 4.2.1 Experimental Setup

We choose two Intel CPUs and two NVidia GPUs for our experiments, and the specifications of the hardware are listed in Table 4.1. The E5-2650v3 and Titan X represent a recent generation of their kind while the E5-2670 and Titan represent high-end hardware that are 3-4 years old. Plus, the corresponding CPU and GPU hardware have very similar price tags. The E5-2630v3 and E5-2670 are installed on two separate servers running Red Hat Linux under kernel version 2.6.32 and GCC version 4.4.7. The GPUs are connected via PCI-E 3.0 16X interface to the same server that hosts the E5-2630v3. Our GPU code is compiled under NVCC 7.5. We also use an NVidia tool named *NVProfiler* to study the runtime characteristics of our GPU code. To maximize the performance of the CPUs, we

run 16 threads for the CPU code, which is the optimal number obtained from a series of tests.

Unless specified otherwise, we set the two input relations to be of the same size. Each tuple in the tables consists of two parts: a 32-bit integer unique key and a 32-bit integer payload that serves as the ID of the tuple. The keys are first generated in order and then shuffled randomly. The keys are uniformly distributed between 1 and table size  $N$ . We perform equi-join on the key, the selectivity of the join condition is set to render one output item per tuple.

We first report results on in-memory join where the data size fits the capacity of GPU memory. We compare our code with existing GPU join algorithms and the latest CPU join code, and go through different factors that potentially affect join performance. Finally, we use the GPU to handle large data that exceeds its memory capacity, and compare its performance with CPU.

## 4.2.2 Experimental Results

### 4.2.2.1 Comparing with Existing GPU Code

Defining the appropriate baseline for such experiments has been surprisingly difficult. After a thorough investigation of the known related work, our comparisons are focused on the GPU join programs presented in He *et al.* [1]. Among the multitude of studies on GPU database systems, few discussed join algorithm design and implementation. Others [22, 23] focus on query engine without clearly modularized code for joins. Another work [12] aims at improving data transmission efficiency by UVA while uses the code of [1] as building blocks. Therefore, we are confident that [1] is by far the most up-to-date and systematic work on GPU-based joins. Plus, their code is also used by CPU-based parallel join work [15] as a comparative baseline. Our attempts to extract and test standalone join code from the work

**Table 4.2:** Resource utilization of major kernels in the new and old GPU sort-merge join code

| Kernel                      | New Algorithms |            | Existing Algorithms |           |
|-----------------------------|----------------|------------|---------------------|-----------|
|                             | BlockSort      | Merge      | partBitonic         | Bitonic   |
| Block Size                  | 256            | 256        | 512                 | 512       |
| Registers/Thread            | 41             | 31         | 16                  | 10        |
| Shared Memory/Block         | 9KB            | 9KB        | 4KB                 | 0KB       |
| Occupancy Achieved          | 62.1%          | 98.8%      | 93.2%               | 84.8%     |
| Shared Memory Bandwidth Use | 3308.2GB/s     | 1098.6GB/s | 1585.9GB/s          | 0GB/s     |
| L2 Cache Bandwidth Use      | 84.6GB/s       | 295.3GB/s  | 110.1GB/s           | 262.6GB/s |
| Global Memory Bandwidth Use | 84.5GB/s       | 253.3GB/s  | 109.5GB/s           | 262.9GB/s |

of [23] and [22] failed due to compilation errors and lack of documentation to help fix such errors.

According to Fig. 4.7, our GPU code significantly outperforms that introduced in [1]. Specifically, the new sort-merge join achieves 4.0-4.9X speedup, with speedup goes slightly higher as the data size increases. On the other hand, a 2.0-14.6X speedup is observed for the new hash join. The same results can be seen in both the Maxwell Titan X and Kepler Titan cards. Only issue is that due to the small global memory of Titan (6GB), the join code cannot run under a 128M table size. The large variation of the speedup in hash join is caused by the partitioning strategy of the old code. In particular, when table size reaches 32 million tuples, the partitioning process changes from two-pass to three-pass in order to keep each partition small. This results in a sudden increase of running time. In contrast to that, the new hash join generates more partitions per pass thus we ensure two passes is enough for a large range of data sizes. As a result, its running time grows proportionally to the input size.

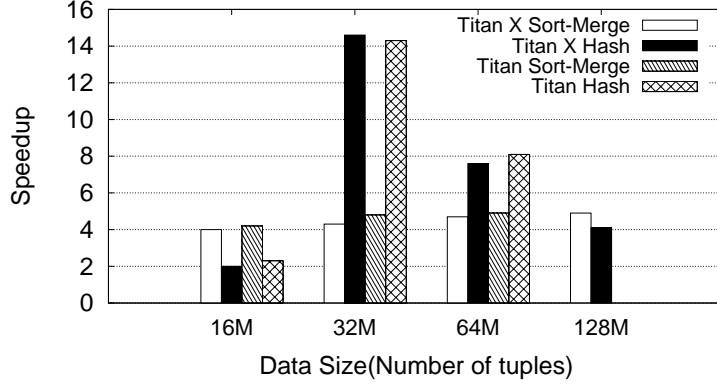
To get insights on the big performance gap between old and new joins, we study the GPU resource utilization achieved by major kernels in both pieces of code. Such data are collected via NVProfler and presented in Tables 4.2 and 4.3. Note the block sizes shown represent those that deliver the best kernel performance. For sort-merge join (Table 4.2), the old code used a bitonic sorting network that directly operates on global memory. Only when

**Table 4.3:** Resource utilization of major kernels in the new and old GPU hash join code

| Kernel                      | New Algorithms |           |           | Existing Algorithms |          |           |
|-----------------------------|----------------|-----------|-----------|---------------------|----------|-----------|
|                             | Histogram      | Reorder   | Probe     | Histogram           | Reorder  | Probe     |
| Block Size                  | 256            | 256       | 256       | 8                   | 8        | 128       |
| Registers/Thread            | 13             | 20        | 22        | 14                  | 16       | 18        |
| Shared Memory/Block         | 4KB            | 4KB       | 4KB       | 8KB                 | 8KB      | 4KB       |
| Occupancy Achieved          | 87.6%          | 89.1%     | 91.0%     | 16.6%               | 16.4%    | 83.1%     |
| Shared Memory Bandwidth Use | 201.5GB/s      | 19.5GB/s  | 775.3GB/s | 275.9GB/s           | 85.6GB/s | 637.3GB/s |
| L2 Cache Bandwidth Use      | 357.3GB/s      | 171.3GB/s | 28.3GB/s  | 36.4GB/s            | 59.8GB/s | 28.6GB/s  |
| Global Memory Bandwidth Use | 103.2GB/s      | 98.1GB/s  | 8.5GB/s   | 36.4GB/s            | 58.9GB/s | 23.3GB/s  |

sorting a partition of the data (kernel *PartBitonic*), the shared memory is used but only 50% bandwidth (1586GB/s) is utilized. When combining all the partitions (kernel *Bitonic*), the accesses to the global memory are entirely random and non-coalesced. Although these kernels have relatively high multiprocessor occupancy (e.g., the number of threads that can run at the same time on a multiprocessor), they are bounded by the utilization of shared memory and bandwidth of global memory, respectively. On contrary, our new sort-merge join makes every step local to the threads. In the *blocksort* kernel, each thread sorts their own items in registers in a sequential manner with zero latency. Then the the whole block of threads combine their tiles together in the shared memory. Even though the occupancy of this kernel is only 62%, the nearly 100% (3.3TB/s) bandwidth utilization on the shared memory ensures the overall performance. Furthermore, all the merging operations are also completed in shared memory. Finally, all the data are in order and can be output to global memory efficiently with coalesced access.

For hash join (Table 4.3), the main problem with the old code is the unbalanced use of GPU resources. In particular, due to the lack of atomic operations in older GPUs, each thread keeps its own copy of an intermediate output (i.e., histogram of radix partition) in the shared memory. As a result, in the *Histogram* and *Reorder* kernels, only eight threads can be put into each block. That is even smaller than the basic scheduling unit of the GPU, which is 32 threads (a warp) at a time. Because of that, only 16% occupancy is achieved by these kernels, meaning that the multiprocessors are extremely underutilized.

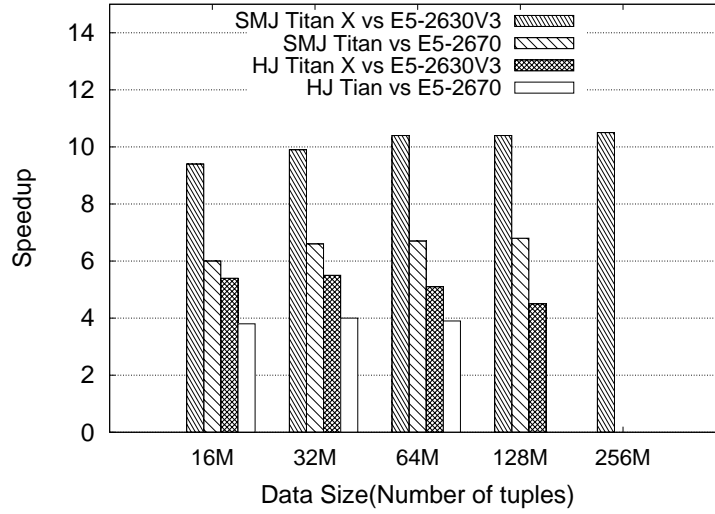


**Figure 4.7:** Speedup of new GPU join algorithms over existing GPU code under different table sizes

In our redesigned hash join kernels, both the histogram kernel and reorder kernel achieve more than 87% occupancy. With the help of atomic operation, one copy of shared histogram is kept for a block, thus only 4KB of shared memory is used even for a block size of 256. Writing to global memory is also improved because of the shared histogram. All threads in a block write to a limited space of the output. This increases locality thus the utilization of L2 cache increases. In both sort-merge and hash joins, use of registers per block has increased significantly to take advantage of the large register file in the latest GPU.

Previous work [15, 20] concluded that hash join is more efficient than sort-merge join in current CPU hardware, while the latter would benefit from wider SIMD instructions. For GPUs, the key to this problem is the utilization of the memory system. The sorting stage in the sort-merge join relies heavily on the fast shared memory and register file to reorganize the inputs. However, the radix partition of the hash join has more random access, thus is hard to be localized into shared memory. At best, the memory access can be cached by L2, but its bandwidth is one magnitude lower than that of shared memory. Therefore, in our code the sort-merge join is up to 26% faster than the hash join.





**Figure 4.8:** Speedup of our GPU code over the latest CPU code

#### 4.2.2.2 Comparing with Latest CPU Code

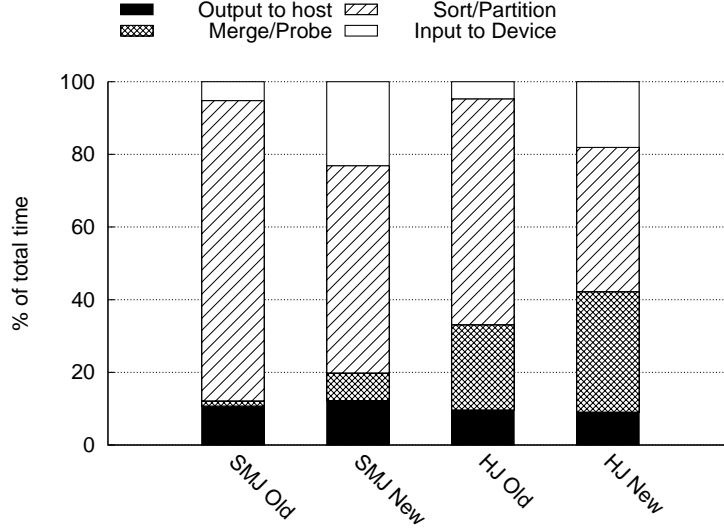
The CPU code we use for our comparisons are developed by Balkesen and co-workers [17, 20], which is obviously the most efficient parallel developments for both sort-merge and hash joins. Fig. 4.8 shows the relative performance of our GPU code to the latest CPU-based joins. We first want to point out that the older E5-2670 outperforms the newer E5-2630v3 in all cases but the newer Titan X GPU is always the winner. Therefore, the relative performance between Titan X and E5-2630v3 shows the maximal GPU-to-CPU speedup while that of Titan to E5-2670 shows the minimal in our tests. Clearly, the GPUs outperform CPUs in both sort-merge join and hash join by a large margin. In sort-merge join, the Maxwell Titan X achieves more than 10X speedup over the Haswell E5-2630V3, while the Kepler Titan has up to 6.8X speedup over the Sandy-Bridge E5-2670. In hash join, the advantage of GPUs shrinks but is still considerable, our code running on Titan X achieves a 5.5X speedup over the E5-2630V3, while the Titan obtains a 4.0X speedup over the E5-2670.

In terms of performance improvement between two generations of hardware, the GPUs see more benefit. The Maxwell Titan X improves by 22% and 35% in overall performance over the Kepler Titan for sort-merge join and hash join, respectively. This can be easily interpreted as the result of the computing capacity of new generations of GPUs that increased significantly over the past few years (Table 4.1). On the CPU side, the newer Haswell E5-2630v3 is even 26% and 2% slower than the older E5-2670 in sort-merge join and hash join, respectively. This shows that the architectural update on CPUs does not bring any performance advantage in join processing. Although the E5-2630v3 works on a new generation of memory (i.e., DDR4), the higher clock rate of E5-2670 cores actually makes better use of the memory bandwidth.

#### 4.2.2.3 Time Breakdown

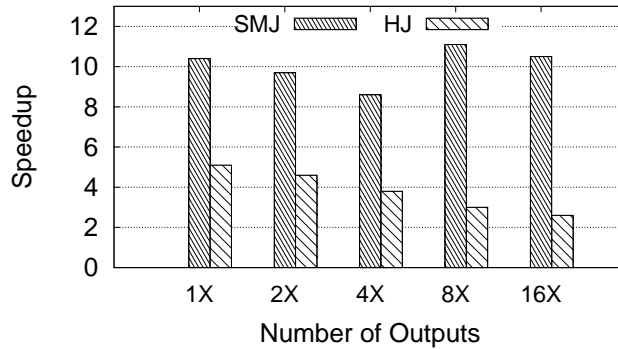
The execution time breakdown of our GPU code and that provided by [1] is shown in Fig. 4.9. The first thing we notice is that the transmission of input/output data to/from GPU is an extra cost for the GPU code, and it counts for 35% and 27% of the total time in the new sort-merge join and hash join, respectively. Since the join kernels of sort merge is faster than hash join, the data transfer time takes up higher percentage in hash join – almost 1/3 – of the total execution time.

When comparing the new algorithms with the old ones, we find that the join processing time in new code contributes less to the total running time while the data transfer time contributes more. In sort-merge join, the percentage of sorting stage time drops from 82.7% to 57.1%, which corresponds to a 7X of performance speedup. The merge-join is, however, not a time consuming stage, taking up less than 8% of execution time. The reason why the merge-join stage in our new code is a little slower is that the old code uses a different mechanism. It builds tree indexes for one of the input relation after sorting. The merge stage gained some benefit from the indexes. But our sort-merge join is still much faster in terms of GPU processing time. In hash join, both partition and probe stages are much



**Figure 4.9:** Execution time breakdown (percentage) of new and old GPU algorithms running on Titan X

faster than existing code, achieving 6.2X and 3.8X speedup respectively. The results indicate that our newly designed kernels are more efficient than those in the existing code by using optimizations that take advantages of the new GPU architectural features. If we do not consider the time for data transfer between host and GPU, both sort-merge and hash in GPU will get a much higher speedup. For sort-merge the speedup would become 15.5-17.5X while for hash join it is 6.3-8.3X. Obviously, a GPU is way more efficient than a CPU in processing the join itself but gets a big hit in data communication via the PCI-E bus.



**Figure 4.10:** Impact of join selectivity on speedup of Titan X over E5-2630v3 under data size 64M

#### 4.2.2.4 *Effects of Join Selectivity*

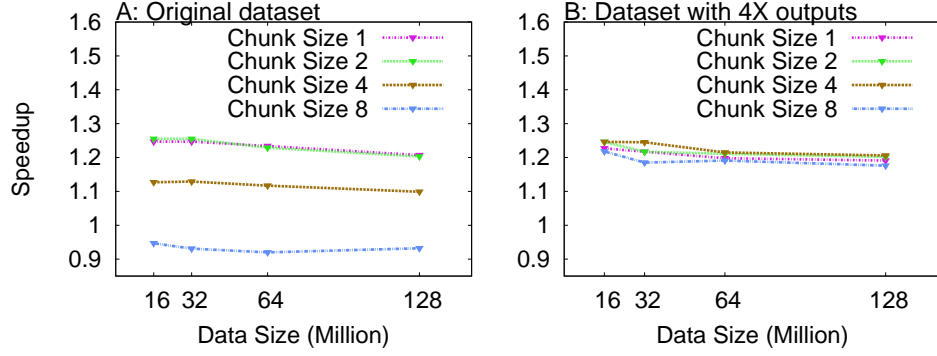
Fig. 4.10 shows the impact of varying selectivity, i.e., the total number of output tuples. The GPU sort-merge join enjoys a speedup around 10X over the CPU except at 4X of outputs where it drops to 8.5X. On the other hand, the GPU hash join suffers from the increasing outputs with a decreasing speedup over the CPU from 5.1X to 2.6X. It is expected that when more tuples are generated as a result of the join, the GPU program will bear a higher overhead as more data will be written back to host via the PCI-E bus. This explains why the hash join performance degrades. However, the impact of selectivity on sort join performance does not seem obvious. By scrutinizing the behavior of our code, we found that the actual running time of our sort merge code does increase as more output tuples are returned. On the other hand, due to a special design of a data structure for holding output tuples, the CPU-based sort-merge join code sees serious performance cut when the output size increases.<sup>8</sup> This overshadows the performance loss observed in GPU code therefore the GPU-to-CPU speedup stays on the same level. As a general trend, we believe lower selectivity will hurt the performance of GPU programs to a extent that there is no competitive advantage of GPUs, as we discussed earlier in 4.2.2.3. But our strategy of overlapping data transmission and join processing can also offset such effects.

#### 4.2.2.5 *Effects of Direct Output*

By using the direct output buffer, the hash join sees a significant benefit. Fig. 4.11A shows the results of our hash join code comparing with the same code without using a direct output buffer. Under page size of one, improvement starts with 25% under 16M data size and, as the input data becomes larger, the improvement gradually drops down to 20%. Such drop is due to the increase of atomic operations to acquire the pointer to the buffer

---

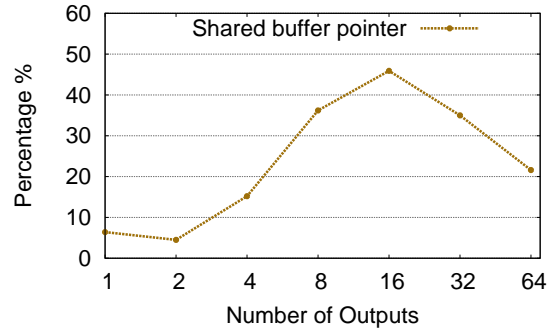
<sup>8</sup>To be fair, this is likely a small problem that can be easily fixed. However, we decided to keep the CPU code as intact as possible for a more accurate comparison.



**Figure 4.11:** Speedup of direct output vs. double probe in the new hash join

in global memory. When the input size increases, the number of output tuples also grows proportionally. Each thread has to request more chunks to store the output, thus increases the number of atomic operations, as a overhead to the code. For the data sizes we tested, the overhead is acceptable. We test this technique with the sort-merge join as well, but it does not improve the performance because the join stage in sort-merge join is different from that in hash join. A linear search is used for the sorted data to determine the range of the output without scanning the whole table, so it saves more time compared with the double-probing approach in the hash join.

We also ran tests to determine the optimal page size for the output buffer. To our surprise, small page sizes of one or two helps achieve the best performance with our original dataset. This is mainly because larger page size also requires larger overall buffer size since there may be empty holes in some of the pages. The time spent on transferring the output buffer back to main memory increases as the result of increasing buffer size. This could offset the benefit of reducing atomic operations. However, larger chunk size may help when the number of outputs per thread increases. Therefore we ran the test on a dataset of the same size as our original dataset but generates 4 times of the outputs, and the result is shown in Fig. 4.11B. As we see that the four different chunk sizes have similar performance at 64M and 128M, while the chunk size four stands out at smaller data sizes. Chunk size of eight is the worst case, indicating that there are still empty holes in it.



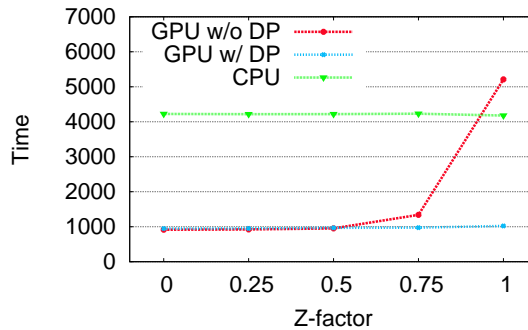
**Figure 4.12:** Performance gained by using shared memory buffer pointer vs. global buffer pointer

We also tested how the buffer chunk size affects the performance when the total number of threads decreases and work per thread increases. Since when outputs per thread increases, a larger page size helps reduce the number of requests to the global pointer. However, the results indicate that larger chunk size only brings marginal improvement. It is possible that the atomic operation in GPU is implemented very efficiently and the pointer is cached in L2, thus the atomic operation is not so sensitive to contentions.

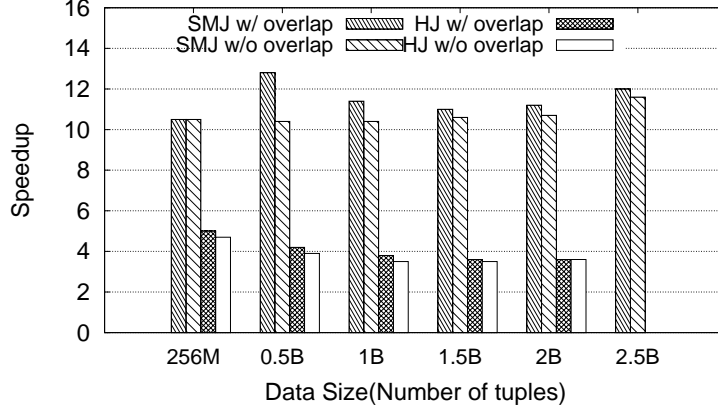
Another way to reduce contention is to distribute the acquisitions of the shared pointer to thread block level. We divide the output buffer into small chunks so that each block can take one of them and outputs independently. The threads in the same block share a pointer in the shared memory that points to the next available slot in their own chunk. A thread acquires the pointer and increase it using atomic operation, then outputs to the available position. Larger selectivity benefits from this method, as shown in Fig. 4.12. Maximum improvement of 45.9% is achieved when the number of output is 16X. However, as the number of outputs continues to increase, the number of atomic operations on shared pointers also comes to a point where it begins to limit performance improvement.

#### 4.2.2.6 Effects of Skewed Data

This section we present the performance of both the CPU and new GPU hash joins when the data has a skewed distribution (in the hashed domain). Specifically, we generate data that follow the Zipf distribution with different  $z$  factors. We run a version of our hash join without the dynamic parallelism (DP) code, and it obviously suffers from imbalance among the partitions under skewed data (Fig. 4.13). As the  $z$ -factor increases, data is more skewed and there is more performance degradation. Particularly, when the  $z$ -factor goes beyond 0.5, only a few blocks are kept busy processing the largest partitions while most of other blocks finish early. In the extreme case of  $z = 1$ , it causes a 4X slowdown as compared to the case of balanced data (i.e.,  $z = 0$ ). After applying DP to the code, threads can determine whether current partition is too large for their thread blocks to process, thus launch additional threads in a child kernel to work only on this partition. The total execution time does not change significantly as the  $z$ -factor increases. However, we do notice that there is a slight penalty when the  $z$ -factor reaches 0.75. This is mainly due to the overhead of launching new kernels. The CPU code is not affected much by data skew. In fact, the CPU code tackles this problem using a similar idea but in a slightly different way. It decomposes unexpectedly large partitions into smaller chunks. The small chunks are processed by using all the threads.



**Figure 4.13:** Performance of CPU/GPU code under different levels of data skewness



**Figure 4.14:** Speedup of Titan X over E5-2630v3 with large tables

#### 4.2.2.7 Joins under Large Data

Now we report the results of using new GPU join algorithms to handle large data that exceeds the capacity of GPU global memory. In such experiments, we keep the size of table  $R$  fixed (128M tuples for hash join and 256M tuples for sort-merge join) and vary the size of table  $S$  from 256M to 2.56 billion tuples. In order to process such a large table, we slice it into chunks and all of the chunks take turns to join with table  $R$ . It is worth mentioning that since the memory usage of hash join is higher than the sort-merge join, hash join can only handle a 128M-tuple chunk at a time while the sort-merge join takes a 256M-tuple chunk in each iteration. So for a given data size, the hash join have to go through more loops which impacts the overall performance.

Fig 4.14 shows the speedup of the Titan X over the E5-2630v3. The sort-merge join on GPUs is more capable of processing large data, resulting in speedup between 11X to 13X. Its speedup fluctuates but does not decrease as the size of table  $S$  increases. Since the GPU sort-merge join algorithm needs fewer loops than the hash join, the running time grows in a nearly linear manner. This is the reason why it maintains the high speedup. The hash join on GPU achieves a 5.1X speedup under 256M tuples. However, it decreases as the table size increases and converges to around 3.5X. The kernel execution and data transfer overlapping



(via multiple CUDA streams) is effective for both algorithms. However, the effects of such are less significant than we thought: on average, there is a performance gain of 8% and 6% for sort-merge join and hash join, respectively. By looking into the profiles of our code, we found that the main reason is that various kernel synchronization activities decrease the level of concurrency at runtime. Note that the CPU hash join code actually sets a limit on table size such that it cannot handle the case of 2.5B records in table  $S$ .

## Chapter 5: Efficient Join Algorithms For Large Tables with Multiple GPUs

Relational join is one of the core components in database management systems (DBMS). It is an essential operator for many database applications that involve multiple tables. Hence improving join processing performance has been an active topic in database research. The increasing database size in today’s business applications has imposed significant challenges to efficient processing of relational joins.

Modern hardware technologies, especially multi-core and many-core chips, provide abundant computing capabilities. As a result, there is a large body of work on join algorithms designed and optimized for such hardware. On multicore CPUs, various strategies [18, 16, 17, 20, 15] such as workload partition and assignment, cache optimization, and use of SIMD instructions are proposed.

Many-core computing platforms, represented by Graphics Processing Units (GPUs), also attracted much attention from the research community. It has been demonstrated [26, 12, 11, 1, 27, 28] that GPUs, as a general-purpose parallel computing platform, are very promising in processing relational joins – a speedup of 5-10X has been reported when comparing GPU join code with multi-thread CPU code. All such work, however, is based on the assumption that *input tables and intermediate join results can be fully loaded to the GPU memory*. In a typical high-end GPU, the size of the on-board memory is at the 8-16GB level, this sets a tight limit on the scale of database that can be processed. In this chapter, we propose efficient algorithms for processing joins when both input tables are too big to be GPU resident.

Nowadays, more and more workstation and servers carry multiple GPU devices. The combined computing and memory storage capability of (four to eight) GPUs in such a

machine can easily dwarf those of a cluster that consists of dozens of nodes seen a few years ago. A main strategy of our algorithm design is to take advantage of the combined capabilities of multiple GPUs in a single node (although our algorithms also work under single-GPU scenarios). To the best of our knowledge, this is the first work that relaxes the “small table” assumption by using multiple GPUs in join processing.

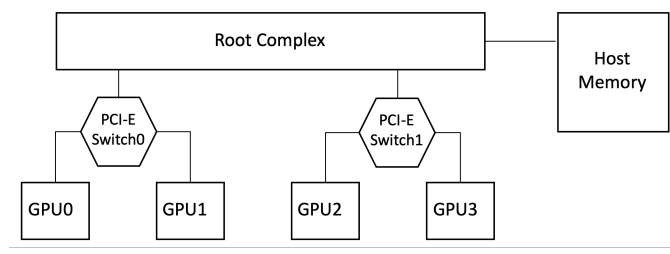
There are a few challenges in using multiple GPUs for large table join processing. First, since the device memory capacity of GPUs is limited, it is inevitable that data exchange frequently occurs between the CPUs and the GPUs via PCI-E bus. The bandwidth of PCI-E is a major bottleneck even though it has increased over the years. To make things worse, multiple GPUs share the same PCI-E channels to communicate with the CPUs that further deteriorates GPU-CPU data exchange rate. Therefore it is vital to minimize the data transfer overhead. Second, in a multi-GPU environment, data sharing among GPUs may be necessary. The inter-GPU communication is also accomplished via PCI-E which may significantly impact the join performance. Although certain patterns of communication can maximize the bandwidth utilization, it creates extra work to share the data that slows down the speed. Hence, the purpose of our algorithm design is to overcome the aforementioned challenges and take full advantage of multi-GPU platform for join processing.

This chapter makes the following contributions. First, by exploring the hardware hierarchy and design space of multi-GPU join algorithms, we identify the main obstacles against efficient large table joins on multiple GPUs. Specifically, we conclude that the low memory transfer rate and complex structure of the data communication links are the main issues, therefore setting the focus of algorithmic design to minimizing the data transfer among GPUs and CPUs. Second, we propose three distinctive designs of multi-GPU join algorithms: nested loop, global sort-merge, and hybrid joins. These algorithms can handle very large input tables, and each has its applicability to joins with different types of conditions. Last, we evaluate the performance of our algorithms against various data tables with sizes

up to 12 billion tuples using two servers featuring PCI-E and NVLink interconnections, respectively. The best of our algorithms show linear scalability on table size, and also achieved much better performance with the use of multiple GPUs (e.g., up to 2.8X speedup for four GPUs compared with a single GPU). Our algorithms significantly outperform multi-thread CPU join code, with a speedup in the range of 6.9-27.5X.

### 5.1 Challenges in Multi-GPU Joins

In modern general-purpose GPU computing systems (e.g., CUDA), the data have to be explicitly transferred from main memory to the GPU’s on-board memory (called *global memory*) for in-core computation. As compared to CPU-based solutions, this is a pure overhead. The overhead is especially significant in join processing because relational joins often involve very little in-core computation (e.g., simple data comparison instead of heavy arithmetic operations). Normally, the running time of a GPU task consists of the following three parts: time  $T_1$  for shipping input data from main memory to GPU global memory, time  $T_2$  for GPU computation, and time  $T_3$  for shipping output data back to CPU. The total time for running a task is therefore  $T_1 + T_2 + T_3$ . In CUDA, an instrument called *CUDA streams* can be used to reduce the overhead by overlapping data transfer with in-core computation. Theoretically, use of CUDA streams can lead to a total running time of  $\max(T_1, T_2, T_3)$ . As shown in previous work [27],  $T_2$  is generally a non-dominant part of total running time. In addition to the light-computation nature of joins, development of optimized GPU join algorithms also made  $T_2$  less significant. This situation is worsened by the imbalanced growth of GPU resources over the years – increase of interconnection bandwidth has lagged behind that of in-core computing capabilities of GPUs.



**Figure 5.1:** A typical system layout of four GPUs connecting to one CPU socket

#### 5.1.1 Limited PCI-E Bandwidth

The use of multiple GPUs in processing large table joins, while further complicating the issue, still sees inter-device communication as the major cost. The limited capacity of a main-stream GPU’s global memory (up to 16GB) only allows it to process a portion of the database tables at a time. Therefore, data exchange among all the computing devices (including CPUs and GPUs) is inevitable and more frequent than the single-GPU with small data scenario. The actual bandwidth of the PCI-E link (at  $32GB/s$  bidirectional) is at most in par with the host memory bandwidth, further restricting fast data transfer. Hence, the overhead of data shipment is more significant when the amount of data becomes larger. As a result, a key aspect of designing GPU join algorithms for larger dataset is to minimize the overhead brought by such data transmission.

Each PCIE connection consists of multiple physical lanes that provide up to  $32GB/s$  bidirectional bandwidth in total. When multiple GPUs need to copy data from/to the host simultaneously, they must share the bandwidth of the only host-to-GPU connection. When the number of GPUs increases from 1 to  $N$ , the combined computing power increases by a factor of  $N$ , but the total host-to-GPU bandwidth remains the same. This also means the host-to-GPU data transfer rate for each GPU is reduced by  $N$ . Hence, it is important to eliminate back-and-forth data transmission between the host and the GPUs in designing join algorithms.

### 5.1.2 Complex Inter-GPU Communication Pattern

In addition to its limited bandwidth, the structure of the PCI-E interconnection introduces extra complexity. If there is only one GPU in the system, the connection is end-to-end and bi-directional between the host memory and the GPU. When multiple GPUs are connected to the system, it forms a tree structure that consists of GPUs and PCI-E switches, as shown in Figure 5.1. Each direct link between two points is bi-directional and has full PCI-E 16X bandwidth. Again, any concurrent traffic passing through the same PCI-E link have to share the bandwidth. Fortunately, the PCI-E links are two-way duplex, meaning it provides bi-directional data exchange capabilities. The number of GPUs supported by one socket can be up to 8, in which case there are two levels of switches and a root complex. Data transmission between GPUs follows the shortest path between the devices, i.e., in Figure 5.1, GPU0 and GPU1 communicate via Switch0, but GPU1 and GPU2 have to follow a path of *Switch0-root-Switch1*. GPU-to-host communications have to go through the root complex.

For systems with multiple CPU sockets, each socket could connect to a network shown above. As a result, the system has more PCI-E lanes for device-device data transfer. However, data movement between sockets have to go through the inter-socket connection (i.e., QPI for Intel and X Bus for IBM), which has much lower bandwidth than in intra-socket transfer. Moreover, the data have to hop to different regions of the host memory before reaching the target GPU, giving rise to large data transfer latency. Such regions are generally called *non-uniform memory access* (NUMA) regions. Hence it is critical to minimize the data exchange between NUMA regions and GPUs that are attached to different CPU sockets.

Note that recent Nvidia cards are equipped with high-speed NVLink interconnection with 80GB/s of bi-directional bandwidth. From our experiments, we observe that the NVLink

**Table 5.1:** Common symbol definitions

| Symbol | Definition                     |
|--------|--------------------------------|
| D      | Number of GPUs                 |
| R, S   | Relations to be joined         |
| M      | Number of chunks in relation R |
| N      | Number of chunks in relation S |
| nRP    | Number of Radix Partitions     |

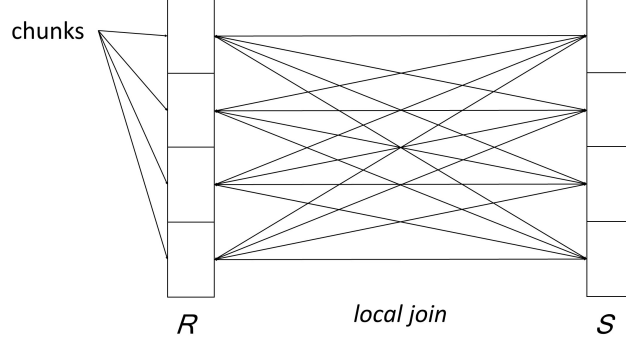
shares similar characteristics with the PCI-E except the former provides higher bandwidth. With NVLink, the data transfer is still a bottleneck in processing joins between large tables.

## 5.2 Large Table Join Algorithms

In this section, we describe three variants of multi-GPU join algorithms in detail. Since the focus of this work is to explore the design of large joins with multiple GPUs by reducing the inter-device data transfer overhead, we utilize single-GPU join algorithms proposed in [27] as the basic building block of our work. We chose such primitives because they so far deliver the best performance (as we have verified via extensive experiments) in a single-GPU-small-table setup. In the following discussions, we assume that both input tables are too big to reside in GPU memory. However, we assume the host main memory is large enough to host both tables. Our algorithm design and implementation are based on NVidia’s CUDA programming framework. Symbols and notations used throughout this chapter are listed in Table 5.1.

### 5.2.1 Block-Based Nested-Loop Join

We start by the intuitive idea of nested-loop join with blocking strategy as the first multi-GPU join algorithm. By blocking, we mean that the join is carried out between blocks (or chunks) of the input tables. This algorithm is convenient for implementation of theta join or for use with data already residing in local partitions. As shown in Figure 5.2, the two input



**Figure 5.2:** Overview of block-based nested-loop join

tables  $R$  and  $S$  are split into equally-sized chunks so that a pair of chunks (one from each of  $R$  and  $S$ ) along with intermediate data can fit into the global memory of a GPU. Upon loading a chunk from the outer table  $R$  to each GPU, all chunks of the inner table  $S$  will be loaded one by one into the same GPU. Depending on what the join conditions, we can use different in-core join algorithms such as nested loop, sort-merge, or hash joins shown in [27] and [1]. Our algorithm terminates when all chunks of  $R$  are consumed. Algorithm 6 shows the pseudocode of the nested-loop join.

---

**Algorithm 6:** Block-Based Nested-Loop Join with in-core Sort-Merge Join

---

**Input:** Relation  $R$ ,  $S$ , Number of GPUs  $D$ , Number of Chunks  $M$ ,  $N$

**Output:** Join result  $res$

```

1:  $r[] = \text{partition}(R, M)$ ;
2:  $s[] = \text{partition}(S, N)$ ;
3: for  $i = 0$  to  $M-1$  omp parallel do
4:    $\text{setCudaDevice}(i\%D)$ ;
5:    $r' = r[i]$ ;
6:   for  $j = i$ ;  $j < N$ ;  $j += D$  do
7:      $s' = s[j]$ 
8:      $res.append(\text{gpuInCoreJoin}(r', s'))$ ; //local join
9:     for  $k = 1$  to  $D-1$  do
10:       $s' = \text{memCopy}(s', (i+1)\%D)$ ; //copy local  $s[i]$  to next device
11:       $res.append(\text{gpuInCoreJoin}(r', s'))$ ;
12:     end for
13:   end for
14: end for

```

---



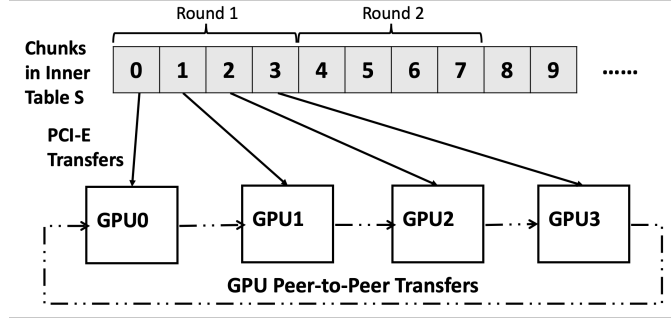
The design of the inner loop (line 11 in Algorithm 1) is worth special consideration. The easiest way would be to directly fetch chunks of  $S$  from the host memory. However, this requires bandwidth sharing in the PCI-E root complex among all GPU cards (Figure 5.1), leading to a major bottleneck. Our strategy is to have the GPUs exchange their chunks of  $S$  (Figure 5.3) before asking for new chunks from the host memory. Recall (Figure 5.1) that peer-to-peer data transmission could be done in lower levels of the PCI-E network tree, thus reducing the load on the high-level switches including the root complex. Given that, the inner loop of Algorithm 1 is divided into stages, in each stage the  $D$  GPUs will each load a chunk of  $S$ , the  $D$  chunks of  $S$  will be consumed by all GPUs before the next stage starts.

An interesting problem here is: *given the  $D$  chunks, how do we arrange the P2P transmission among  $D$  GPUs such that time for every GPU to see all chunks is minimized?* With the PCI-E network structure represented in Figure 5.1, our solution is as follows: in a PCI-E network of  $D$  GPUs, the  $i$ -th GPU ships its data to its immediate neighbor, e.g., the  $(i + 1) \% D$ -th GPU (Figure 5.3). By this, it takes  $D - 1$  rounds to share all the data, and we will take full advantage of the data channels in the entire PCI-E network. We can prove that with the PCI-E network structure represented in Fig. 5.1, the Ring exchange plan requires the smallest amount of time to exchange all data for all GPUs, though the proof is omitted here due to the article length.

In all algorithms presented in this chapter, by “omp parallel” we mean to unroll the loop and distribute the workload (via the loop index) to different GPUs via OpenMP.

#### 5.2.1.1 Performance Analysis

Suppose there are a total of  $M$  chunks in table  $R$  and  $N$  chunks in table  $S$ , the number of devices is  $D$  and the PCI-E uni-directional bandwidth is  $B$ . Intuitively, without P2P data



**Figure 5.3:** Loading of inner table chunks and GPU peer-to-peer data transmission in the nested-loop join

transmission, the total amount of data transferred between the host and the devices is

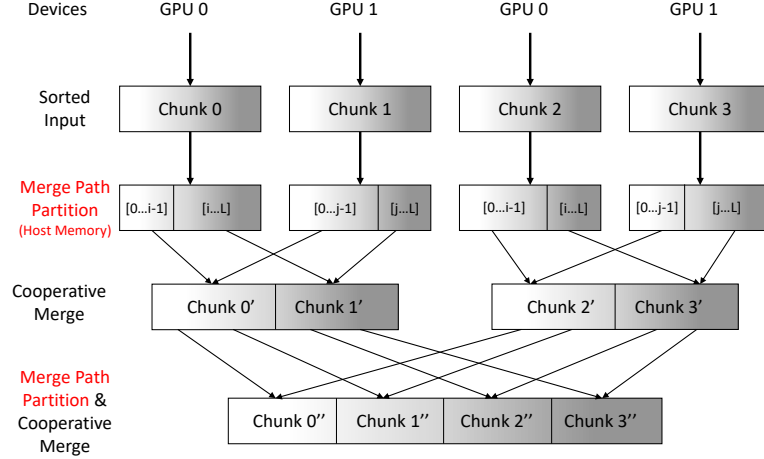
$$\mathcal{T}_1 = |R| + M|S| \quad (5.1)$$

where  $|\cdot|$  is the size of a table. By taking advantage of the PCI-E P2P transfer, we reduce the amount of data transferred between the host and the devices to  $|R| + \frac{M}{D}|S|$ . In addition to that, we have to add the data traffic via P2P transfer. For each step of the inner loop, there are  $D$  chunks of  $S$  that need to be exchanged, and there are  $D(D-1)$  P2P exchanges to be done. Thus, the total amount of P2P data transfer is  $\frac{M}{D} \cdot \frac{|S|}{D} D(D-1) = \frac{M|S|}{D}(D-1)$ . Given those, the amount of data transfer of the whole algorithm (with P2P) is

$$\mathcal{T}_2 = |R| + \frac{M}{D}|S| + \frac{M|S|}{D}(D-1) \quad (5.2)$$

However, to compare the time needed to accomplish the transfer of  $\mathcal{T}_1$  and  $\mathcal{T}_2$ , we need to apply a  $D$ -fold discount to the 3rd item of Eq. (5.2), and the bandwidth-adjusted  $\mathcal{T}_2$  is

$$\mathcal{T}_2' = |R| + \frac{M}{D}|S| + \frac{M|S|}{D^2}(D-1) = |R| + M|S| \left( \frac{2D-1}{D^2} \right) \quad (5.3)$$



**Figure 5.4:** Global sorting with two GPUs

By comparing Eq. (5.3) with Eq. (5.1), we can clearly see a performance advantage brought by the P2P data transfer. By ignoring the low-order item  $|R|$  in both equations, the improvement is by a factor of  $D^2/(2D - 1)$ .

### 5.2.2 Global Sort-Merge Join

While the nested-loop join can handle arbitrary join conditions, it carries a quadratic data transmission cost. For joins with more special conditions such as equality or range matches, we introduce our second join algorithm named global sort-merge join. It is based on the well-known idea of sort-merge join which brings the data transmission cost down to subquadratic. The key challenge of it would be to design a sorting algorithm that works with multiple GPUs and is able to consume input larger than the total GPU memory size. It is not trivial to do so given the complexity of workload assignment, load-balancing and data sharing issues with a multi-GPU setup.

In traditional disk-based DBMS, sorting depends on serial merge of two sorted lists, each of which has to be accomplished by one worker thread/CPU. For GPUs, we can do the same, but it is less efficient since GPUs cannot cooperate on merging a pair of large sorted

lists, leaving some of them idling. In the in-memory setup, the random-access host memory allows for CPUs to search through the input and partition the workload for multiple GPUs. Therefore we developed a multi-GPU sorting algorithm that features CPU-assisted partition that enables multi-GPU merge.

The global sort-merge algorithm consists of two steps: sort and join. In order to realize the out-of-core processing and multi-GPU parallelism, the sort step is further split in to two parts: sorted-run generation and multi-GPU merge. The join step applies the same idea of multi-GPU merge to conduct multiple in-core joins. With the CPU-assisted merge path partitioning, each GPU works on a pair of disjoint partitions of the two tables and multiple GPU devices are capable of working independently and cooperatively. We elaborate each step in the following sections.

#### *5.2.2.1 Sorted Run Generation*

Since we assume the input tables can be arbitrarily large and the GPUs' global memory capacity is limited, divide-and-conquer is necessary. Therefore the sorting must be split into two parts: generating small sorted runs and merging the sorted runs into one sorted table. The sorted run generation ensures that the sorted runs are small enough to fit in a GPU's global memory along with its intermediate data structures, and each sorted run can be processed locally by a GPU. As shown in Algorithm 7 and Figure 5.4, the input relations are split into equal-sized chunks whose sizes fit in the global memory of a single GPU. Each of the GPUs takes one chunk at a time from the host memory with a fixed stride. Then it uses efficient in-core GPU radix sort algorithm [45] to sort the chunk locally, and transfer it back to the host memory.

A key challenge is to hide the latency of data transfer via the shared PCI-E channels. For that, we use three CUDA streams for each GPU to overlap computation and data transfer. Each stream performs the complete host-to-device transfer, sorting and device-

to-host transfer process as a pipeline. Therefore the three pipelines can overlap each other so that the compute resources and PCI-E bandwidth are kept busy most of the time. To facilitate the execution of the pipelines, the available global memory space on the GPU is divided into three parts. So each stream has its own work space, there is no contention among the streams when accessing or transferring data. This applies to all three steps in this algorithm and the third algorithm which we will introduce in Section 5.2.3. All GPUs work together in parallel until all the chunks are sorted.

#### 5.2.2.2 *Multi-GPU Merge of Sorted Runs*

When the sorted runs are ready, they need to be merged into larger sorted lists. However, it is impossible for a GPU to hold two small sorted runs in its global memory. Multiple GPUs must work cooperatively when merging the sorted runs and perform several rounds of merging if the sublists to be merged are larger than the total size of all GPUs' global memory. Therefore a data partition is necessary to ensure parallelism and load balancing among the GPUs. To tackle this challenge, we propose a multi-threaded CPU-assisted merge-path partition [41, 42] and multi-GPU merge, the core mechanism that enables out-of-core processing with multiple GPUs, which is shown in Algorithm 8. Merge path is a parallel partition algorithm that utilizes binary search on two sorted lists to partition them into load-balanced workload for multi-threaded merging. It breaks the merge workload of two sorted lists  $A$  and  $B$  into multiple equal-sized portions by using binary search along the pairs  $A[i]$  and  $B[j]$ , where  $i + j = L$  and  $L$  is the partition size.

In each round of merge, multiple threads are launched using OpenMP in accordance with the number of GPUs  $D$  so that there is one CPU thread working with a GPU. Before the merge begins, the threads apply merge path partition on the two sublists to be merged. Each GPU works on a pair of partitions independently. Since the merge path partition is a binary search into both sorted lists to ensure the merges are load-balanced, the GPUs all

have the same amount of work even though the partitions may not be of the same size. The GPUs fetch their partitions from the host memory and write the merged list back to the host memory afterwards. If the size of the pair of sublists to be merged are too large, they will be split into more partitions so that multiple GPUs can work cooperatively on them. If there is only one GPU available, it can process the partitions as well.

Figure 5.4 illustrates the workflow with an example of two GPUs and four input chunks. After GPU 0 and 1 take turns to sort the chunks, the data are all transferred back to the host. Each of chunks 0-3 is partitioned by CPU into two partitions using merge path in host memory (labeled by red font). GPU0 first works on merging elements 0 to  $i-1$  of chunk 0 and elements 0 to  $j-1$  of chunk 1, and then merging the corresponding partitions of chunks 2 and 3. Meanwhile, GPU1 first works on merging elements  $i$  to  $L$  of chunk 0 and elements  $j$  to  $L$  of chunk 1, and then merging the other partitions of chunks 2 and 3. As a result, chunk 0' and 1' are generated and form a single sorted list, while chunk 2' and 3' form another one. Then the two newly generated sorted lists go through another round of partitioning and merging by CPUs and GPUs respectively, until the four chunks are merged into a single sorted list.

### 5.2.2.3 Multi-GPU Merge Join

As both input tables are sorted and ready in the host memory, we can proceed to the merge join stage (Algorithm 9). Similar to the idea of multi-GPU merge in previous step, we launch  $D$  CPU threads by OpenMP and use merge path partition to split both tables into  $D$  partitions. Each GPU acquires a pair of partitions from  $R$  and  $S$  and join its own pair independently. It is possible that the size of each pair of partitions still exceeds a GPU's global memory capacity. Therefore for each GPU we apply the merge path partition again that splits each pair of partitions into even smaller ones so that one GPU can join a small pair of partitions using the in-core merge join algorithm within the limit of its global memory

size. Each GPU joins the smaller partitions in a sequential manner until it finishes the whole pair of partitions.

#### 5.2.2.4 Performance Analysis

In the sorted run generation, we still assume there are  $M$  sorted runs in  $R$  and  $N$  in  $S$  for simplicity. The total data transferred between the host and GPUs including copying sorted runs back to main memory is  $2(|R| + |S|)$ . The multi-GPU merge stage requires  $\log_2 M$  and  $\log_2 N$  rounds to merge all the sorted runs in  $R$  and  $S$  respectively. Therefore the total data transferred between host and GPUs is  $2(|R|\log_2 M + |S|\log_2 N)$ . During the above two steps, we take advantage of CUDA streams and bi-directional data transfer, so the transfer bandwidth is  $2B$ . In the final merge join, the data traffic is  $|R| + |S|$ . In summary, the total amount of data transmitted between host and GPUs is

$$3(|R| + |S|) + 2(|R|\log_2 M + |S|\log_2 N) \quad (5.4)$$

As a result, the time spent on data transfer over PCI-E is

$$\frac{2(|R| + |S|)}{B} + \frac{|R|\log_2 M + |S|\log_2 N}{B} \quad (5.5)$$

As compared to the Nested-Loop Join, the advantage of the Global Sort-Merge Join lies in its low I/O cost. It was achieved by the global sorting such that only those chunks of  $R$  and  $S$  that could generate join results will be processed by the GPUs for the final in-core merge.

### 5.2.3 Hybrid Join

Although the global sort-merge join is handy when dealing with equi-join and range join, it comes with a price of transferring data back and forth to accomplish the sorting task.

---

**Algorithm 7:** Global Sort-Merge Join

---

**Input:** Relation  $R$ ,  $S$ , Number of GPUs  $D$ , Number of Chunks  $M$ ,  $N$

**Output:** Join result  $res$

```
1:  $r[] = \text{partition}(R, M)$ ;  
2:  $s[] = \text{partition}(S, N)$ ;  
3: for  $i = 0$  to  $M-1$  omp parallel do  
4:    $\text{setCudaDevice}(i\%D)$ ;  
5:    $\text{gpuSort}(r[i])$ ;  
6: end for  
7:  $\text{cooperativeMerge}(r, M, D)$ ;  
8: for  $i = 0$  to  $N-1$  omp parallel do  
9:    $\text{setCudaDevice}(i\%D)$ ;  
10:   $\text{gpuSort}(s[i])$ ;  
11: end for  
12:  $\text{cooperativeMerge}(s, N, D)$ ;  
13:  $res = \text{globalJoin}(r, s, D)$ ;
```

---

To further reduce the data transfer, we need a linear time partitioning or partial sorting mechanism. Our previous work [27] on GPU-based hash and sort-merge joins has revealed that fact *the in-core sort-merge join is more efficient than the radix-partitioned hash join on GPUs*. Another key observation is: *the radix partition stage of the hash join is faster than the sorting stage of the sort-merge join*, which implies we can use radix partitioning to further reduce the I/O costs of the Global Sort-Merge Join. That leads to the idea of a *hybrid* design that enjoys the merits of both worlds - a combination of global radix partitioning and in-core sort-merge join. This new design features out-of-core multi-GPU radix partition and in-core sort-merge join on individual GPUs (see details in Algorithm 10), while tackles the same challenges of multi-GPU environments. In this algorithm, we first group the tuples with the same radix value in each of the input tables into small partitions, then each GPU takes a pair of partitions from  $R$  and  $S$  and proceed to in-core sort-merge joins independently. Both steps are scalable with more GPUs. We want to point out that, due to the global hashing, this algorithm only works for equality joins.



---

**Algorithm 8:** cooperativeMerge

---

**Input:** Partitioned Relation  $r$ , Number of Chunks  $M$ , Number of GPUs  $D$

**Output:** Sorted Relation  $r$

```
1: stride = 2;
2: numPairs = M/stride;
3: for  $i = 0$  to numPairs-1 do
4:   for  $k = 0$  to stride-1 omp parallel do
5:     setCudaDevice( $i\%D$ );
6:     cpuMergePath( $r[2*k]$ ,  $r[2*k+1]$ );
7:     gpuMerge( $r[2*k]$ ,  $r[2*k+1]$ );
8:   end for
9: end for
```

---

---

**Algorithm 9:** globalJoin

---

**Input:** Sorted Relation  $R$ ,  $S$ , Number of GPUs  $D$ , Max Join Size  $T$

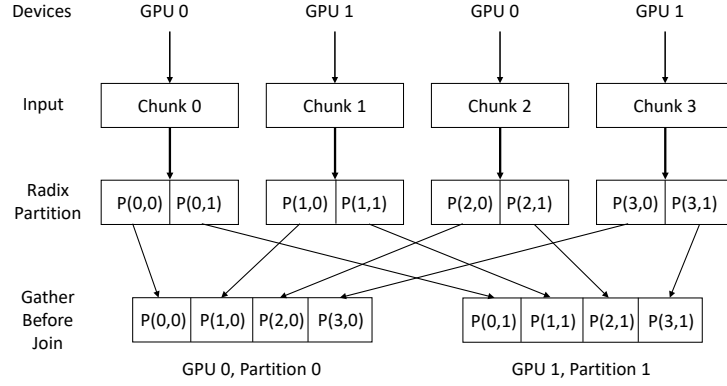
**Output:** Join Results  $res$

```
1: [numPartitions,  $r[]$ ,  $s[]$ ] = mergePathPartition( $R$ ,  $S$ ,  $T$ );
2: for  $i = 0$  to numPartitions-1 omp parallel do
3:   setCudaDevice( $i\%D$ );
4:    $res +=$  inCoreSMJ( $r[i]$ ,  $s[i]$ );
5: end for
```

---

### 5.2.3.1 Multi-GPU Radix Partition

The first step of hybrid join is to partition the input tables. Unlike other partitioning strategies, radix partitioning creates disjoint partitions that are suitable for parallel sort-merge join tasks since one partition in one table has only one corresponding partition in the other table, hence reducing computation and memory access cost. In addition, extracting the radix value only requires bit operations which are more efficient than the comparison-based mechanism. The radix partition consists of three steps: building histogram, computing prefix-sum and scattering tuples to new positions. Building histogram is to count the number of tuples falling in each bucket that represents a particular radix value. A prefix scan on the histogram results in prefix-sum, an array of numbers that can be used to determine the starting position to scatter the tuples in each bucket. Based on the prefix-sum, we can



**Figure 5.5:** Radix partition with two GPUs

---

**Algorithm 10:** Hybrid Join

---

**Input:** Relation  $R$ ,  $S$ , Number of Chunks  $M$ ,  $N$ , Number of Radix Partitions  $nRP$ , Number of GPUs  $D$

**Output:** Join result  $res$

```

1:  $r[] = \text{partition}(R, M)$ ;
2:  $s[] = \text{partition}(S, N)$ ;
3:  $\text{presumR}[M][nRP] = \text{gpuRadixPartition}(r[], M, D)$ ;
4:  $\text{presumS}[N][nRP] = \text{gpuRadixPartition}(s[], N, D)$ ;
5: for  $i = 0$  to  $nRP$  omp parallel do
6:    $\text{setCudaDevice}(i\%D)$ ;
7:    $r' = \text{gather}(r[], \text{presumR}[], M, i)$ ;
8:    $s' = \text{gather}(s[], \text{presumS}[], N, i)$ ;
9:    $\text{gpuSort}(r')$ ;
10:   $\text{gpuSort}(s')$ ;
11:   $res.append(\text{inCoreSMJ}(r', s'))$ ;
12: end for

```

---

reorder the tuples in parallel. Previous work addressed the radix partition on multi-core CPUs and on single GPUs [15, 1, 27]. In this section, we focus on our design with multiple GPUs.

To make the GPUs work simultaneously, we need to distribute the input among them and then combine their local histograms. Algorithm 11 describes the workflow in detail. To build the histogram, each input table is scanned once. Therefore each GPU simply fetch a chunk of the input that fits in the global memory at a time. Then the GPUs proceed to in-core histogram kernel for each chunk, and keep an array of the partial histograms of every thread block in the GPU’s global memory. Every time a GPU processes a new chunk, it counts the partial histograms for that particular chunk accordingly. The GPUs keep fetching chunks and computing histograms until the input is exhausted. Upon the completion of counting histogram, each chunk have its own partial histogram on the data.

There are two options for the following prefix-sum and scatter. The intuitive way is to combine the partial histograms from all the chunks, consequently generating a global prefix sum and the tuples are scattered to different contiguous spaces (or buckets). However it has a few drawbacks. On the contrary, the other option allows for lower overhead by letting the GPUs build local prefix-sum within each chunk. By doing this, the radix partition is done at chunk level rather than the whole table.

Right after the histogram is built for a particular chunk, the corresponding GPU begins to calculate its local prefix-sum and then scatters the tuples to the buckets in GPU’s memory space. Therefore, it is not necessary to keep the histogram of that particular chunk afterwards, reducing the memory consumption and extra data transfer overhead. It also helps reduce scatter overhead since it can be done within the global memory, eliminating the need for UVA. The GPUs keep working chunk by chunk until all of them are processed. The only minor issue with this method is that the tuples belonging to the same bucket spread in several different chunks although they are clustered locally within each chunk. In the in-

core join stage, the separated sub-partitions of each partition have to be fetched separately. Since we have the prefix-sums for all the chunks that help us decide the starting and ending positions of the sub-partitions, the problem should be easily solved by copying them one by one.

---

**Algorithm 11:** gpuRadixPartition

---

**Input:** Partitions of Relation  $r$ , Number of Chunks  $M$ ,  
Number of Radix Partitions  $nRP$ , Number of GPUs  $D$   
**Output:** prefix of the radix partitions of all the chunks  $globalHisto$

- 1: **for**  $i = 0$  to  $M-1$  **omp parallel do**
- 2:   setCudaDevice( $i\%D$ );
- 3:   histogram[ $nRP$ ] = gpuComputeHistogram( $r[i]$ ,  $nRP$ );
- 4:   presum[ $nRP$ ] = gpuComputePrefix(histogram);
- 5:   gpuReorderRelation( $r[i]$ ,presum);
- 6:   globalHisto.append(presum[]);
- 7: **end for**

---

The workflow of the multi-GPU radix partition is shown in Figure 5.5, where an example of two GPUs and four chunks is used. For GPU 0, it takes chunk 0 of table R at first from the main memory and perform radix partition on it (we use two partitions  $P(0,0)$  and  $P(0,1)$  for illustration purpose), then puts the reordered chunk back to main memory. The two GPUs partition the chunks in stride manner until all the chunks are partitioned. The GPUs do the same to the chunks of table S, we omit this step for simplicity. The GPUs gathers the data of the same partition from different chunks before the in-core join starts. In order to achieve higher work efficiency, we use three CUDA streams to pipeline the partitioning process, overlapping the data transfer with the actual computation consisting of histogram, prefix-scan and scatter.

### 5.2.3.2 In-Core Sort-Merge Join

After the radix partition, the paired partitions with the same radix value from the two tables can be joined on the GPUs independently. As mentioned above, the tuples of a

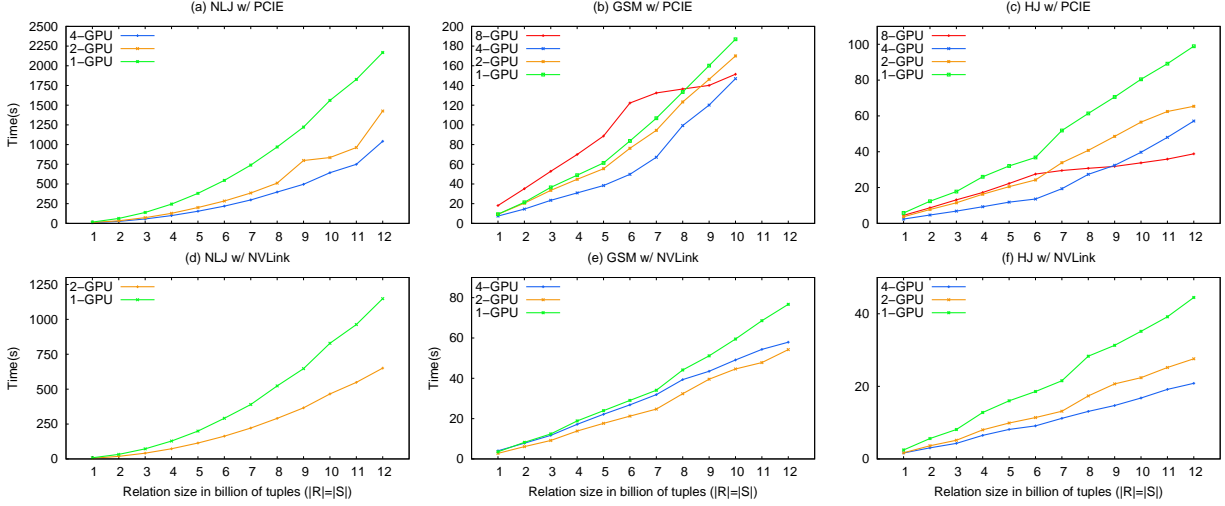
particular partition are scattered into many sub-partitions across multiple chunks. Hence at the beginning of the in-core join, the sub-partitions need to be gathered as a whole partition, as shown in Figure 5.5. To do so, we iterate over all the chunks and fetch the corresponding sub-partitions with the particular radix value one at a time with the help of the partial prefix-sum generated in the previous stage. The sub-partitions are copied into a buffer in the GPUs' global memory. In the example, GPU 0 is assigned to gather  $P(0,0)$  through  $P(3,0)$  since they all belong to partition 0. As soon as the input partitions are ready in the global memory, the GPUs perform in-core sort-merge join in parallel. The join results are buffered in another array in the global memory. When the in-core join procedure finishes, the results are transferred to the main memory and combined into one list. To further accelerate the in-core join stage, we pipeline the process using three CUDA streams to overlap the join processing with data transfer.

### 5.2.3.3 Performance Analysis

In the hybrid join approach, data transfers occur at radix partitioning stage and only involve a linear scan of the input from the main memory and writing back to the main memory. The in-core join stage only reads the reordered input once. Therefore, the total data traffic over PCI-E is

$$3(|R| + |S|). \quad (5.6)$$

Due to the fact that the scan and write in the radix partitioning stage can be overlapped using bidirectional bandwidth of the PCI-E channels, the data transfer time of that stage is



**Figure 5.6:** Running time of three join algorithms with PCIE and NVLink

reduced by half.<sup>9</sup> Thus, the total data transfer time is

$$\frac{2(|R| + |S|)}{B} \quad (5.7)$$

The above shows that the hybrid join is clearly superior to the other algorithms introduced earlier.

### 5.3 Experiments

To evaluate the performance and scalability of three join algorithms, we test them on two different hardware platforms. The first one consists of two Intel Xeon E5-2630V3 CPUs with 384GB of RAM, where each Xeon processor socket connects four Nvidia GTX Titan X Pascal GPUs via PCI-E3.0 16X. The second one is an IBM Minsky server with two Power8 processors, 512GB of RAM, and each processor socket connects two Nvidia Tesla P100 GPUs via NVLink. Both platforms have CUDA 9.0 installed on Linux operating systems. Each

<sup>9</sup>The actual data transfer rate may be different. For example, our tests show that we can achieve 12GB/s uni-directional and 22GB/s bi-directional. Thus, the data transfer time will reduce by a factor of  $1-12/22=0.45$ , which is slightly lower than 0.5.

**Table 5.2:** Speedup of multi-GPU runs of different algorithms over single-GPU runs on GTX Titan X pascal

|         | Hybrid Join |           |           | Global SMJ |           |           | Nested-Loop Join |           |
|---------|-------------|-----------|-----------|------------|-----------|-----------|------------------|-----------|
|         | 8 GPU       | 4 GPU     | 2 GPU     | 8 GPU      | 4 GPU     | 2 GPU     | 4 GPU            | 2 GPU     |
| Average | 1.81        | 2.38      | 1.52      | 0.81       | 1.47      | 1.08      | 2.40             | 1.81      |
| Range   | 1.30–2.55   | 1.73–2.80 | 1.42–1.60 | 0.52–1.23  | 1.26–1.68 | 1.01–1.13 | 2.06–2.49        | 1.52–1.92 |

**Table 5.3:** Speedup of multi-GPU runs of different algorithms over single-GPU runs on Tesla P100

|         | Hybrid Join    |                | Global SMJ     |                | Nested-Loop    |
|---------|----------------|----------------|----------------|----------------|----------------|
|         | 4 GPU          | 2 GPU          | 4 GPU          | 2 GPU          | 2 GPU          |
| Average | 1.97           | 1.58           | 1.12           | 1.36           | 1.76           |
| Range   | 1.50<br>– 2.16 | 1.47<br>– 1.64 | 0.88<br>– 1.32 | 1.29<br>– 1.43 | 1.73<br>– 1.80 |

GTX Titan X Pascal has 12GB GDDR5X RAM, while the Tesla P100 has 16GB HBM2 RAM.

We follow the convention found in many existing work [12, 11, 1] on relational joins in generating synthetic data: the tuples are  $\langle key, value \rangle$  pairs where both the key and value are 32-bit integers. The keys are randomly generated following an uniform distribution, and for the tests against skewed data, a Zipf distribution. Unless specified otherwise, we set the table size  $|R|$  and  $|S|$  to be the same.

The parameters M and N (number of chunks) of all algorithms are chosen in a way such that the largest chunks of R and S are used as allowed by the specific algorithms, and we keep the same chunk sizes for both R and S. In particular, the chunk size for the nested-loop join is 250M records, 50M records for global sort-merge and hybrid joins.

For benchmarking purposes, we compare our code with the state-of-the-art parallel CPU hash join code [20] with multi-threading, SIMD and cache-conscious optimization techniques, and the GPU code in [28] which addressed out-of-core join with hash join and a single GPU.

### 5.3.1 Total Running Time

Figure 5.6 shows the end-to-end running time of the three GPU join algorithms under various input sizes and hardware configurations. The top three parts in Figure 5.6 show the result of Titan X pascal with PCI-E interconnections. Undoubtedly, the nested-loop join is the slowest among them, requiring more than 2,000 seconds to join two 12-billion-record tables with one GPU as shown in Figure 5.6(a). Using more GPUs does improve the performance significantly. The improvement margin shrinks as the number of GPUs increases.

The global sort-merge join is about one order of magnitude faster than the nested-loop join. As shown in Figure 5.6(b), the curve is less steep than the quadratic growth of nested-loop join. When using two and four GPUs, the improvement over one GPU is noticeable but not as significant as in the nested-loop join. As we will see in section 5.3.1.2, the reason for this is that the global sort-merge join has a lower amount of work relative to its amount of data transferred, hence the performance is bottlenecked mainly by the saturated bandwidth. Due to the larger size of intermediate results, the global sort-merge join can only handle tables with a size up to 10 billion records.

Obviously, the hybrid join is the winner among the three, outperforming the global sort-merge join by around 2X, as shown in 5.6(c). The multi-GPU speedup is more scalable than global sort-merge join. Although it suffers from the same issue as the latter, the performance of 8-GPU surpasses that of 4-GPU from table size of 9 billion, taking only less than 40 seconds to process the 12-billion-tuple tables.

For the Global Sort Merge and Hybrid joins, we also run experiments under eight GPUs for the PCI-E-based system, with each group of four GPUs connected to one CPU. Note that each such group of four GPUs represent a PCI-E network structure shown in Figure 5.1. Data transfer across the two groups is accomplished via Intel’s QPI link with low bandwidth



and large latency. Therefore, it is expected that the use of GPUs across the QPI link will lower the performance. In our case, after the memory in the first node is used up, as shown at a table size of 6 billions along the red line in Figure 5.6(b), GPUs in both nodes need to access the other node’s memory space. At this point, the QPI’s bidirectional bandwidth can be utilized and the running time curve of 8-GPU becomes less steep. We were not able to run the Nested-Loop Join under eight cards because a key idea of the algorithm, the P2P data sharing, only works for cards connected by one socket.

The bottom three plots in Figure 5.6 show the results of Tesla P100 with NVLink interconnections. The curves exhibit simpler trends than those of the Titan X pascal. The running time grows proportionally across the three algorithms and various number of GPUs used, with the nested-loop join being the slowest and the hybrid join the fastest again. On average, a single Tesla P100 achieves a speedup of 1.9X, 2.9X and 2.2X over a Titan X pascal in the nested loop, global and hybrid joins, respectively. The cases of four GPUs in the Tesla P100 experiments are similar to those of the eight GPUs in the Titan X - they both are connected to two CPUs. As a result, the running time of the global sort-merge join under four P100 is higher than in two GPUs. However, the situation is better in the hybrid join - the performance under four GPU is better than under two GPU setups.

### 5.3.1.1 Scalability

The scalability of a parallel join algorithm is one of the key factors we study. First, let us study the scalability over data size. The running time of the nested-loop join clearly grows in a quadratic manner. The increase of running time for the other two algorithms follows a pattern that is almost linear. There is no significant difference between the growth patterns of these two algorithms.

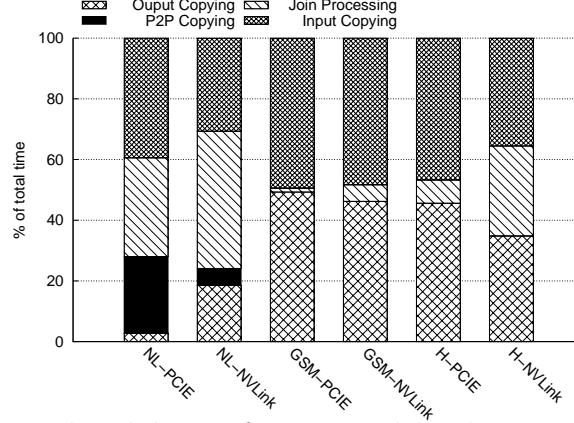
Second, we look at the scalability of the join algorithms over different numbers of GPU devices. In terms of computing power, the performance theoretically grows linear with

the number of GPUs used. However, the data transfer bandwidth and the host memory bandwidth do not change (for both PCI-E and NVLink systems). The impacts of shared bandwidth on performance are as follows.

Tables 5.2 and 5.3 show the speedup of multiple GPUs versus a single GPU achieved in running the three join algorithms on GTX Titan X pascal and Tesla P100, respectively. The nested-loop join algorithm is the least affected by the bandwidth due to two factors. First, its in-core computation carries a heavier weight as compared to the other two variants, minimizing the impact of data transfer (Section 5.3.1.2). Second, the nested-loop join takes advantage of P2P data transfer, effectively multiplying the available bandwidth even though all the data ultimately comes from host memory. As a result, the performance of two GPUs almost doubles the performance of one GPU with x86 and PCI-E, while it shrinks slightly to 1.7X with Power8 and NVlink. Four GPUs working together on PCI-E further improves the performance to 2.4X.

The hybrid join suffers more from the bandwidth issue but is able to maintain an adequate scalability. With two GPUs, the hybrid join achieves more than 1.5x speedup on both x86 and Power8 platforms. The four Titan X on the x86 achieves 2.4X speedup, the same as in the nested-loop join. Both platforms begin to suffer from additional overhead incurred by the cross-die communication when more GPUs are used, resulting in speedup under 2.0X with eight GPUs on the x86 and four GPUs on the Power8. This is because the in-core computation load contributes less to the total running time compared to the nested-loop join, leading to a higher impact of the data transfer bandwidth.

The global sort-merge join was impacted the most by bandwidth in terms of scalability on both testbeds. Even the GPUs attached to the same CPU node cannot work together towards high efficiency. The maximum speedup only reaches 1.5X with four Titan X on the x86. In the global sort-merge join, we need to transfer the data back and forth between the host memory and the GPUs in order to sort and merge the sublists of the input tables. These



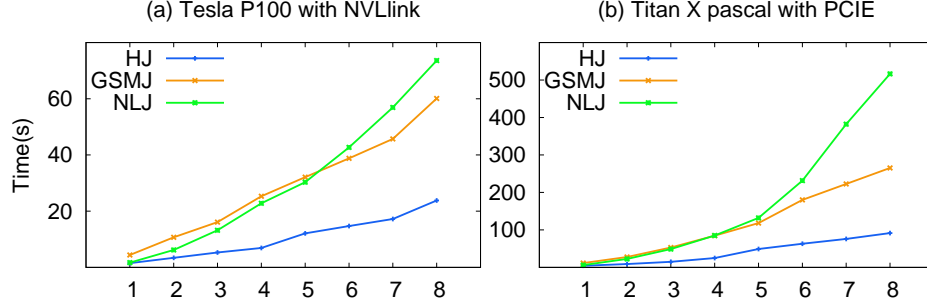
**Figure 5.7:** Running time breakdown of our join algorithms under 8-billion records and 4 GPUs

data transfers have to share the same bandwidth that does not increase with the number of GPUs. The hybrid join, on the other hand, only transfers the data a constant number of times therefore it is less affected.

#### 5.3.1.2 Running Time Breakdown

Figure 5.7 shows the time breakdown of the three join algorithms. The statistics are gathered from a CUDA tool named *nvprof*. Note that we report time measured for each activity (i.e., shipping input data to GPUs, in-core join processing, shipping output data to host memory, and P2P transfer between GPUs) of our algorithms. With the overlapping among such activities via CUDA streams, the sum of all time values reported here will be larger than the total running time reported in 5.3.1.1.

We first look at the difference between PCI-E and NVLink. In all three join variants, the proportion of time spent on data transfer is noticeably higher with PCI-E than that with NVLink. This indicates that the high bandwidth of NVLink directly impacts the performance given that the computing capabilities of the Tesla P100 and GTX Titan X Pascal are roughly the same.



**Figure 5.8:** Data transfer time of the three join algorithms

Then we look at the difference among the three join variants. The nested-loop join spends 30% and 40% of the total execution time on join processing with PCI-E and NVLink respectively, while the other two variants have lower percentage. This is due to the increased amount of work for nested-loop join. Although it has the lowest percentage on data transfer, the absolute data transfer time is still much longer than the other two algorithms. The global sort-merge join spends more than 90% of the time on data transfer as a result of sorting the whole input arrays, since sorting consumes more bandwidth than computing resources. The hybrid join has a higher percentage on join processing than the global sort-merge join because it has linear data transfer time.

Figure 5.8 shows the data transfer time measured by disabling the in-core join processing code. The experimental results validate our cost analysis for the three algorithms when comparing them side-by-side. The data transmission time of nested-loop join grows quadratically as data size increases since the number of chunks  $M$  also depends on data size, while that of global sort-merge join has a flatter curve due to the data transfer reduction by global sorting algorithm. The running time of hybrid join grows linearly thanks to the one pass radix-partition.

### 5.3.1.3 The Effects of CUDA Stream Pipeline

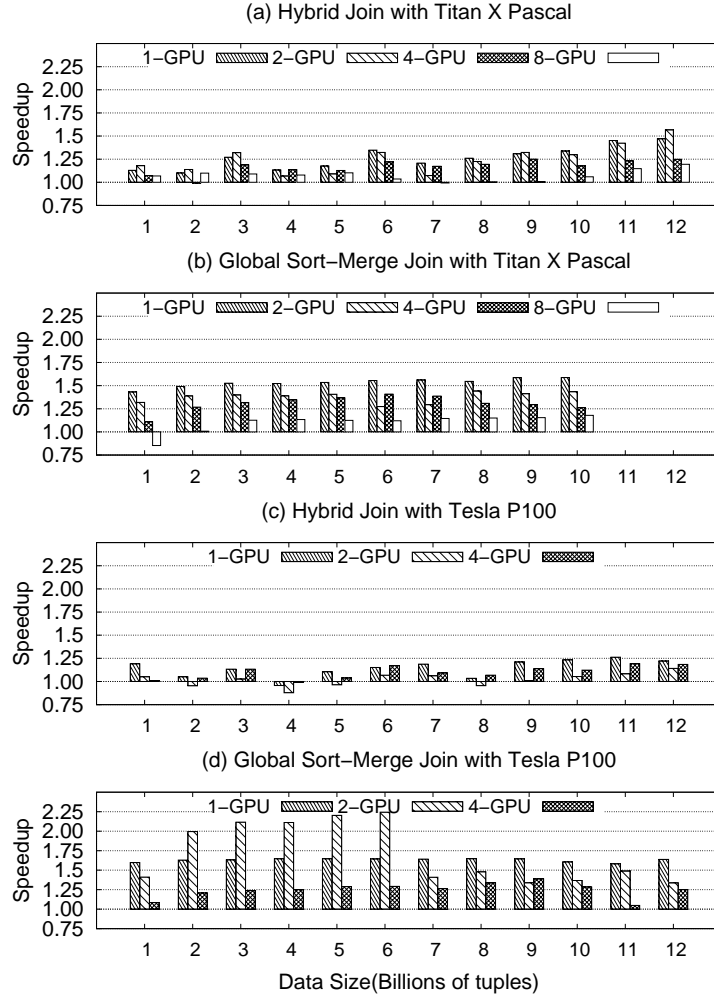
Figures 5.9 shows how the 3-stream pipeline affects the performance of the join performance. In (b) and (d), the bars show the speedup of using three CUDA streams over using one stream in the global sort-merge join under different number of GPUs on both systems. When only one GPU is used for the global sort-merge join, more streams bring a decent 1.5X speedup in both test platforms. With more GPU used, the two platforms both benefit less from more streams with one exception where two Tesla P100 rise to more than 2X from 2 billion to 6 billion of data size, where the GPUs access local host memory region rather than other NUMA regions. In Figures (a) and (c), the hybrid join shares the same overall trend with the global sort-merge join but with much lower speedup and more fluctuations. Using more GPUs does not always benefit from multiple streams. In (rare) cases, streams even deteriorate the performance on the Power8 platform.

By running the code with CUDA profiler, we found that the maximum data transfer speed achieved by asynchronous data transfer is about 10% lower than synchronous data transfer, indicating that asynchronous operations incurs considerable overhead. It turns out that some of the CUDA API calls (e.g., memory allocation/release) are serialized by the system even though the host code is multi-threaded. Besides, the inevitable memory allocations and deallocations also incur more synchronizations, since we cannot reuse the buffers due to the variation in input and output sizes among iterations.

## 5.3.2 Effects of Other Factors

### 5.3.2.1 $|R| : |S|$ Ratio

To investigate how different  $|R| : |S|$  ratios affects the join performance, we conduct experiments where  $|R| + |S|$  is fixed to 16 billion and four GPUs are used wherever possible (except for nested-loop join with NVLink). Figure 5.10 shows the relative running time



**Figure 5.9:** Speedup of 3-stream pipeline

fluctuation when  $|R|$  varies from one billion to seven billion. The nested-loop join and hybrid join take longer to run as  $|R|$  increases. However, the sort-merge join has different behaviors on NVLink and PCIE platforms. With PCIE, the running time are higher on both ends as a result of combined effects of input data transfer cost and output cost. With NVLink, the four GPUs reside in two separate CPU nodes which causes fluctuation in running time.

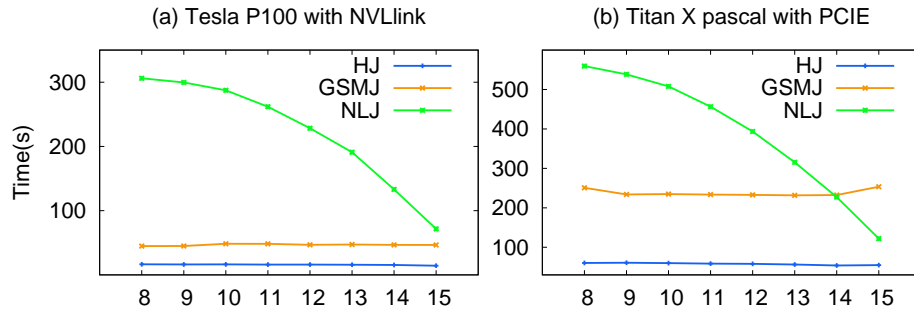
According to the data transfer model in Section 5.2, it affects the three algorithms in different ways. Given that  $|R| + |S|$  is fixed, we can assume there are  $Z$  total pages in  $R$  and  $S$ . For nested-loop join and global sort-merge join, the input data transfer costs are only related to  $(Z - M)M$  and  $M \log_2 M + (Z - M) \log_2 (Z - M)$ , respectively. Therefore, the nested-loop join reaches maximum cost while the global sort-merge join reaches minimum cost when  $|R| = |S|$ . The hybrid join is not affected by this variation. On the other hand, as  $|R|$  increases until it equals to  $|S|$ , the number of output also increases, impacting running time as well. As a result, the nested-loop join and hybrid join should observe longer running time while the global sort-merge join have a complicated running time curve.

#### 5.3.2.2 Selectivity

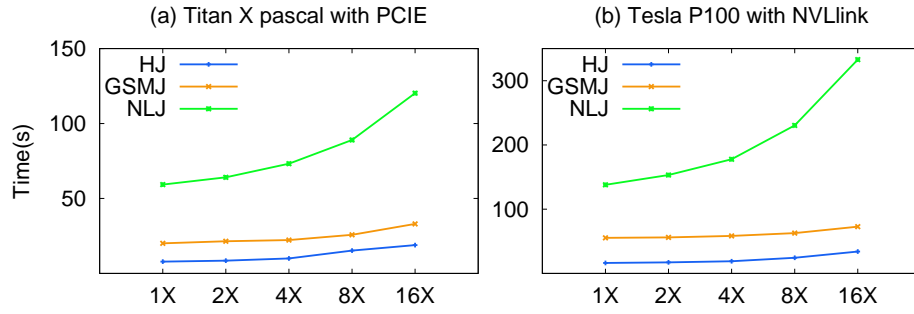
Selectivity determines how many tuples are selected relative to the input size, thus affecting the output size of the join. We test the effect of selectivity by varying the output ratio from 1X to 16X, and the results are shown in Figure 5.11. It is clear that the number of output significantly affects the running time of all the join algorithms. However, the nested-loop join is the most affected by percentage due to its running time surging quadratically. The global sort-merge join and hybrid join have similar trends.

#### 5.3.2.3 Data Skewness

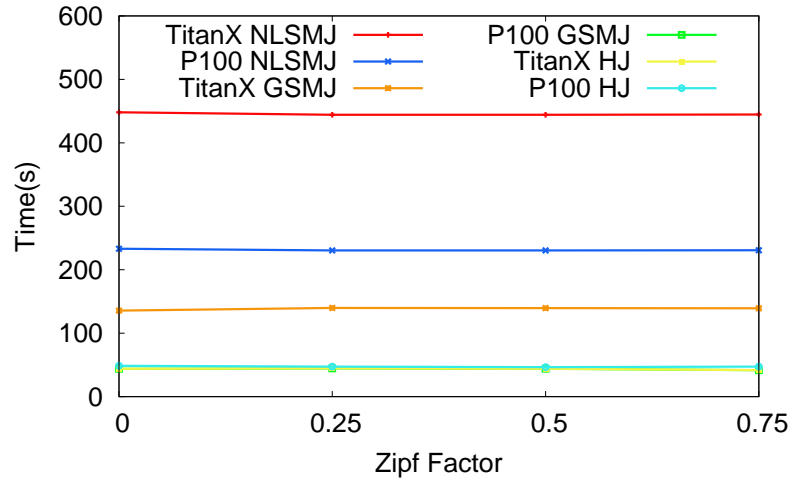
We run our out-of-core GPU join algorithms against skewed data generated following the well-known Zipf distribution. Specifically, we generate join key for one of the tables with



**Figure 5.10:** Running time of various  $|R| : |S|$  ratio

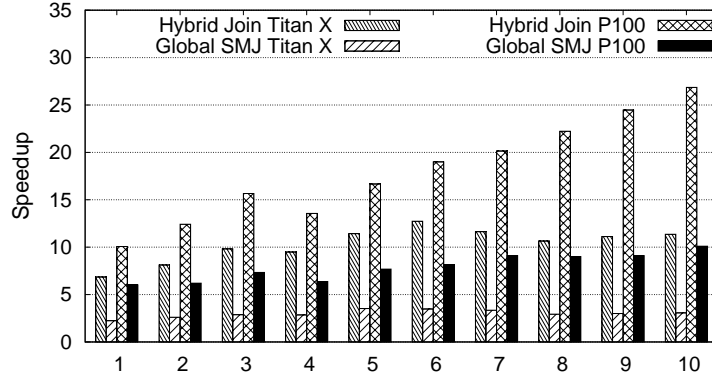


**Figure 5.11:** Running time of various selectivity



**Figure 5.12:** Running time of three GPU join algorithms against skewed data





**Figure 5.13:** The speedup of our out-of-core GPU join algorithms over a CPU-based hash join algorithm

a Zipf factor ranging from 0 to 0.75. The test uses four GPUs and tables of eight billion rows. The result is shown in Figure 5.12, where we can see that none of the algorithms are significantly affected by the skewness of the data.

The nested-loop join takes same-sized chunks in each iteration, while the global-sort-merge join utilizes merge-path partitions. Therefore these two algorithms are naturally load-balanced. The only possible issue would be in the hybrid join, where some of the radix-based partitions are considerably larger than others, potentially resulting in imbalanced loads. However, it turns out that the way that the GPUs handling the partitions actually helps deal with data skew. First, the radix partition uses the least significant bits, and creates fewer large coarse-grained partitions rather than many small fine-grained partitions for the GPUs to work on. As a result, the partitions are less different in size. Second, the GPUs take input partitions in a round-robin manner, hence each GPU works on both larger and smaller partitions, reducing the workload gap among the GPUs. Therefore the overall performance is less affected. By examining the runtime statistics, we notice that the workload difference among the GPUs is only about 10% at most in multiple runs.

### 5.3.3 Speedup Over The State-of-The-Art CPU and GPU Algorithms

We first compare the performance of the global sort-merge join and hybrid join algorithms we proposed with the CPU-based parallel join code presented in [20]. This code is arguably the most efficient CPU-based join code, and is used to benchmark GPU-based solutions in recent studies [27, 28]. For both the CPU and GPU running time, we choose the best results among different setup (number of threads/GPUs) and report the speedup as shown in Figure 5.13. To ensure fair comparisons, we again use the end-to-end running time for GPU algorithms. Specifically, the CPU join algorithm uses eight threads running on the Xeon E5-2640V4. For the GPU algorithms, four Titan X Pascal are used on the x86 platform, while two Tesla P100 are used on the Power8 platform except for the hybrid join algorithm where four Tesla P100 perform the best.

From Figure 5.13, we can see that in most cases the GPU algorithms outperform the CPU hash join algorithm. The global sort-merge join observes a speedup of 2.2-3.1X and 6-10X over the CPU on x86 and Power8, respectively, and its speedup increases as the data size grows. The hybrid join is of the best performance among all the GPU and CPU join algorithms tested, resulting in 6.9-12.7X and 10.1-27.0X speedup with PCI-E and NVLink over the CPU respectively. We also compared the nested-loop join with its CPU counterpart. However, the CPU code takes days to run even with the smallest test case.

From the results, it is obvious that data shipment overhead is still dominant in the performance of the GPU join algorithms. The difference can be as much as 3X if we compare the global sort-merge join with the hybrid join. Previous work has shown that one mid-range GPU is more capable in join processing than a workstation level multi-core CPU. For example, the best GPU-to-CPU speedup for small table joins was reported in [27], with a range of 5.5-10.5X. Our work apparently enlarges the performance gap by using multiple GPUs.

To compare the running time with the state-of-the-art GPU out-of-core join algorithm, we also run the code in [28] with the same setup above. This code features CPU-based multi-threaded radix partitioning and a single GPU hash join. Unfortunately, the code was only able to run up to 3B tuples before the GPU runs out of memory. This is caused by a major constraint of their algorithm, which depends on a hardcoded number of partitions resulting in partitions with sizes larger than the global memory. In terms of performance, our hybrid join algorithm is 11% faster with one GPU at 3 billion and 2.9X faster with four GPUs respectively. We believe with the multi-GPU radix partitioning and efficient in-core sort-merge join, our code would scale better in larger data size.

#### 5.3.4 Discussions

There have been long debates over *which join algorithm is the best*. The sort-merge join and hash join are both commonly used while the hash join is asymptotically faster. In terms of parallel join processing, the answer for the aforementioned question is more complicated since it depends on the specific hardware architecture and the parallel algorithm design and optimization. In general, the algorithm that utilizes the most stringent hardware resources more effectively would perform better. Based on our experiments, the winner is the hybrid join with significantly shorter running time than the other two algorithms. Such results confirm the theoretical bandwidth consumption analysis in Section 5.2, which shows that the hybrid join is superior. However, this by no means makes the nested-loop join and global sort-merge join obsolete. As we have mentioned, the nested-loop join can handle joins with arbitrarily complex conditions. The global sort-merge join is suitable for joins with equality or range match join conditions. The hybrid join can only process joins with equality conditions. One other advantage of the global sort-merge join is that the resulting table is ordered in a particular way. This could lower the costs of downstream operators such as “order by” or “group by” in a database query, and is a frequently utilized mechanism in

query optimization. Given the above facts, all three algorithms will find their uses in an actual GPU-based DBMS.

## Chapter 6: Conclusions

GPGPU is a capable parallel computing platform that also shows its potential in processing database operations. To take advantage of its architectural design for massively parallel computing, join algorithms were developed in previous work and enjoyed up to 7X speedup over CPUs. We revisit the performance of such algorithms on the latest GPU devices to provide an updated evaluation of the suitability of GPGPU in join processing. Our results indicate a significantly expanded performance gap between GPU and CPU, with a GPU-to-CPU speedup up to 20X. By exploiting new hardware/software features such as extra data cache and shared registers, we further boost the performance of chosen algorithms by 30-50%. Upon investigating the floating point performance, energy consumption, and program development issues, we believe GPGPU has also become a mature platform for database operations than before.

We also propose new GPU-based hash join and sort-merge join algorithms. We take advantage of the various new features in the latest GPU hardware and CUDA software. On one hand, it achieves considerable performance boost over the existing state-of-the-art algorithm. The kernels have improved in many aspects including work efficiency and bandwidth utilization. On the other hand, experiments show that our optimized GPU code far outperforms the latest CPU hash join and sort-merge join code. This indicates that the GPU is a promising platform for join processing. Of course, the performance advantage of GPU is not only brought by raw computing power, but also by carefully designed algorithms towards the GPU hardware's features.

We discuss the issues of designing multi-GPU join algorithms to exploit the computing power and minimize the overhead of multi-GPU environments. In particular, we argue that the major bottleneck in a multi-GPU environment in order to fully utilize the capability of all the GPUs is to reduce the time spent on data transfer since the GPUs nowadays have tremendous computing power but the bandwidth is limited. To that end, we designed three join algorithms with different out-of-core data transfer patterns and the same in-core join processing. By a series of experiments, we demonstrated that the inter-GPU communication and data exchange drastically affect the running time of the join algorithms. The peer-to-peer access of PCI-E devices does not help improve the performance although it enables utilization bidirectional bandwidth. Using multiple GPUs also increased join processing performance. The multi-GPU join algorithms achieved up to 2.8X speedup when using up to eight GPUs versus using just one GPU, and up to 27X speedup versus multi-core CPUs. The scalability is mainly limited by the relatively low bandwidth of PCI-E. Therefore the hybrid join with the lowest data traffic among the three algorithms we proposed achieved the best performance. Moreover, we investigate several factors that affect the performance of the join algorithms, including global memory capacity, PCI-E peer-to-peer access, and CUDA streams.

We can extend the scope of the existing large table join algorithms along several dimensions. The first one is to reduce the overall data traffic among GPUs and CPUs by using indexes instead of transmitting the whole tables. This involves the design of indexed join algorithms and efficient index structures on GPUs with optimization towards multi-GPU environments. The second part is to extend the multi-GPU joins to larger scales such as GPU-enabled clusters. There are extra challenges in such environments as the network structures and specifications are different from PCI-E/NVLink, and novel designs are needed to reduce the data transmission overhead. Another thrust of research can be seen at the integration of our algorithms into a real GPU-based DBMS. This requires research into mul-

tiple table joins, various join conditions, interaction with other relational operators, and functions/handles from the join algorithms to assist system-level query optimization.

## References

- [1] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Procs. of SIGMOD*, pages 511–524, 2008.
- [2] TOP 500 List. <http://www.top500.org/lists/2014/06>.
- [3] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration for spatial selections and joins. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 455–466, 2003.
- [4] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *Procs. of VLDB*, pages 1021–1032, 2004.
- [5] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Procs. of SIGMOD*, pages 215–226, 2004.
- [6] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Procs. of SIGMOD*, pages 325–336, 2006.
- [7] Unified Shader Model. [http://en.wikipedia.org/wiki/Unified\\_shader\\_model](http://en.wikipedia.org/wiki/Unified_shader_model).



- [8] CUDA parallel computing platform. [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html).
- [9] OpenCL. <https://www.khronos.org/opencvl>.
- [10] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07*, pages 46:1–46:12, 2007.
- [11] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.
- [12] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Procs. DaMoN*, pages 55–62, 2012.
- [13] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Procs. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10*, pages 94–103, 2010.
- [14] Jiong He, Mian Lu, and Bingsheng He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endowment*, 6(10):889–900, August 2013.
- [15] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nandathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009.
- [16] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Procs. of SIGMOD*, pages 37–48, 2011.

- [17] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.
- [18] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.
- [19] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4):709–730, Jul 2002.
- [20] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013.
- [21] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, December 2014.
- [22] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endowment*, 6(10):817–828, August 2013.
- [23] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Procs. CGO*, pages 44:44–44:54, 2014.
- [24] Yi-Cheng Tu, Anand Kumar, Di Yu, Ran Rui, and Ryan Wheeler. Data Management Systems on GPUs: Promises and Challenges. In *SSDBM*, pages 33:1–33:4, 2013.








- [25] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.
- [26] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *Big Data*, pages 2541–2550, 2015.
- [27] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, SSDBM '17*, pages 17:1–17:12, New York, NY, USA, 2017. ACM.
- [28] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, April 2019.
- [29] J. Paul, B. He, S. Lu, and C. T. Lau. Revisiting hash join on graphics processors: A decade later. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 294–299, April 2019.
- [30] Yongpeng Zhang and F. Mueller. GStream: A General-Purpose Data Streaming Framework on GPU Clusters. In *Procs. 2011 International Conference on Parallel Processing (ICPP)*, pages 245–254, Sept 2011.
- [31] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Procs. VLDB Endowment*, 4(5):314–325, February 2011.
- [32] Peripheral Component Interconnect Express. [http://en.wikipedia.org/wiki/PCI\\_Express](http://en.wikipedia.org/wiki/PCI_Express).


- [33] NVIDIA Tesla. <http://www.nvidia.com/object/tesla-workstations.html>.
- [34] Ran Rui, Hao Li, and Yi-Cheng Tu. Performance Analysis of Join Algorithms on GPUs. Technical Report CSE/14-016, Department of Computer Science and Engineering, University of South Florida, 2014.
- [35] Watts Up Power Meters. <https://www.wattsupmeters.com/secure/products.php?pn=0>.
- [36] Maxwell: The Most Advanced CUDA GPU Ever Made. <http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made>.
- [37] Nvidia Kepler GK110 Architecture Whitepaper. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf>.
- [38] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, December 2000.
- [39] NVIDIA’s Next-Gen Pascal GPU Architecture to Provide 10X Speedup for Deep Learning Apps. <https://blogs.nvidia.com/blog/2015/03/17/pascal/>.
- [40] Evangelia A. Sitaridi and Kenneth A. Ross. Ameliorating Memory Contention of OLAP Operators on GPU Processors. In *DaMoN*, pages 39–47, 2012.
- [41] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *IPDPSW*, pages 1611–1618, 2012.
- [42] Oded Green, Robert McColl, and David A. Bader. GPU Merge Path: A GPU Merging Algorithm. In *Procs of ICS*, pages 331–340, 2012.
- [43] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Procs. 1991 ACM/IEEE Conference on Supercomputing, SC ’91*, pages 712–721, 1991.

- [44] Gpu gems 2, chapter 46, Mar 2005. [http://http.developer.nvidia.com/GPUGems2/gpugems2\\_chapter46.html](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html).
- [45] Nvlab. CUB. <http://https://nvlabs.github.io/cub/>.

## Appendix A: Copyright Permissions

Chapter 3 of this dissertation was published in 2015 IEEE International Conference on Big Data, pp. 2541-2550, doi: 10.1109/BigData.2015.7364051. According to IEEE author rights, the IEEE does not require individuals working on a thesis or dissertation to obtain a formal reuse license.

HomeHelpEmail SupportSign inCreate Account



**Requesting permission to reuse content from an IEEE publication**

**Join algorithms on GPUs: A revisit after seven years**

Conference Proceedings: 2015 IEEE International Conference on Big Data (Big Data)

Author: Ran Rui

Publisher: IEEE

Date: Oct. 2015

Copyright © 2015, IEEE

**Thesis / Dissertation Reuse**

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

*Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:*

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

*Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:*

- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to [http://www.ieee.org/publications\\_standards/publications/rights/rights\\_link.html](http://www.ieee.org/publications_standards/publications/rights/rights_link.html) to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACKCLOSE WINDOW

Chapter 4 of this dissertation was published in Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17). Association for Computing Machinery, New York, NY, USA, Article 17, 1–12. doi: 10.1145/3085504.3085521. According to ACM author rights, Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included.

[authors.acm.org/author-resources/author-rights](https://authors.acm.org/author-resources/author-rights)

### Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is not the editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

### Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.


Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

### Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

 Association for Computing Machinery

About ACM  
About ACM  
Volunteer

Publications  
About Publications  
Digital Library  
Submit a Paper