Revisiting Hash Join on Graphics Processors: A Decade Later

Johns Paul[1], Bingsheng He[2], Shengliang Lu[2], and Chiew Tong Lau[1]

[1]Nanyang Technological University
[2]National University of Singapore

***Abstract*—The large number of computational cores and the high memory bandwidth provided by modern graphics processors (GPUs) make them an ideal hardware accelerator for in-memory hash joins. Over the last decade, significant research effort has been put into improving the performance of hash join operation on GPUs. Over the same period, there have been significant changes to the GPU architecture. Hence, a systematic revisit from the perspective of GPU hardware changes is necessary to understand the past research and to guide future studies.**

In this paper, we first revisit the major GPU hash join implementations in the last decade and detail how they take advantage of different GPU architecture features. We then perform a comprehensive performance evaluation of these implementations using the latest hardware. This helps to shed light on the impact of different architecture features and to identify the factors guiding the choice of these features. Finally, we study how data characteristics like skew and match rate impact the performance of GPU hash join implementations and propose techniques to improve the performance of existing implementations under such conditions.

***Index Terms*—GPU, hash join, partitioning**

I. Introduction

The high level of parallelism and the high memory bandwidth provided by GPUs make them an attractive hardware accelerator for Online Analytical Processing (OLAP). Over the last decade, tremendous amount of research effort has been put into accelerating relational operators like hash join using GPUs [1]–[9]. During this time, in addition to the incremental improvements in the level of parallelism and memory bandwidth, GPU hardware has made major improvements through the introduction of a number of new architecture features. These improvements have made GPUs one of the most promising pieces of hardware for accelerating hash join operation, which is one of the most expensive operators in modern query processing systems.

Hence, we need to revisit GPU hash join implementations in conjunction with the GPU architecture improvements as shown in the timeline in Figure 1. The timeline clearly shows the following trend: every 1-2 years vendors like NVIDIA introduce new architecture features and 2-3 years down the line we see new GPU hash join implementations that outperforms previous implementations by taking advantage of these new features. However, existing literature lacks a systematic revisit of GPU hash join implementations from this perspective. Further, given the trend in Figure 1, understanding the factors guiding the choice of different features is critical to future research and system designs. Hence, in this paper we revisit the hash join implementations in the last decade to understand the impact of different GPU architecture features and to identify the factors guiding the choice of different architecture features.

GPUs are designed to process huge amounts of data in parallel with minimal inter-thread communication. Hence, GPUs are often inefficient when processing unbalanced workloads like skewed input relations. Other GPU hardware limitations like limited global memory size make it difficult to join data sets with high match rate due to possible overflow of output buffers. Still, there is a lack of studies on the impact of such workloads on the overall performance of hash join implementations. To address this, we study the impact of data characteristics like skew and match rate on the performance of hash join implementations and propose techniques to improve their performance for such data sets.

To summarize, the major contributions of this paper are as follows. First, we revisit the major hash join implementations in the last decade and detail how these implementations take advantage of new GPU architecture features. Second, we study the impact of these architecture features on the performance of the hash join operation and identify the factors guiding the choice of different features. Third, we study how data characteristics like skew and high match rate impact the performance of GPU hash join operation and propose techniques to improve the performance of existing implementations when joining such data sets.

II. Background

In this section, we present the background on GPU hardware and detail some key new GPU architecture features. Note that, we mainly focus on NVIDIA GPUs, and leave AMD GPUs as future work.

A. GPU Architecture & CUDA

A single GPU consists of multiple *streaming multiprocessors* (SMs), each of which consists of multiple CUDA cores. Each CUDA core has access to its own set of registers (local memory). Further, each SM contains an L1/texture cache and a shared memory. All the SMs in the GPU share an L2 cache and a global memory. Finally, all data needs to be copied from the CPU main memory to the GPU global memory (via the PCIe bus) before it can be processed by the GPU.

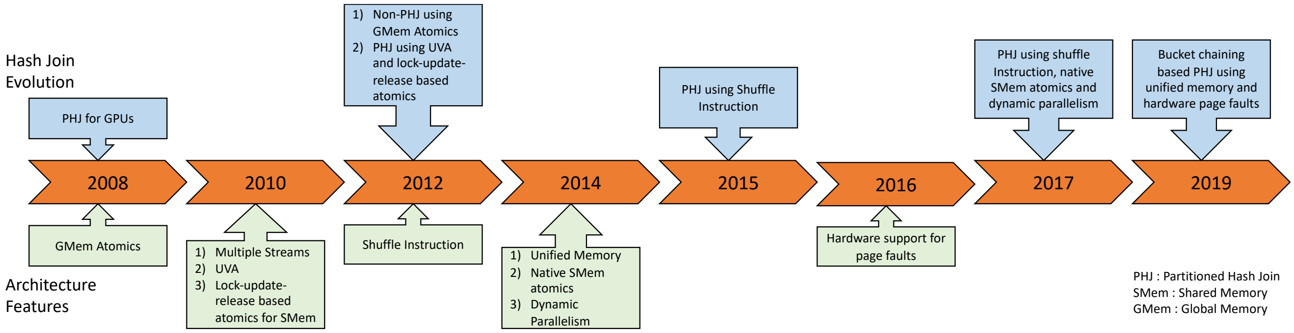
Hash Join Evolution
PHJ for GPUs
1) Non-PHJ using GMem Atomics
2) PHJ using UVA and lock-update-release based atomics
PHJ using Shuffle Instruction
PHJ using shuffle Instruction, native SMem atomics and dynamic parallelism
Bucket chaining based PHJ using unified memory and hardware page faults
2008
2010
2012
2014
2015
2016
2017
2019
GMem Atomics
1) Multiple Streams
2) UVA
3) Lock-update-release based atomics for SMem
Shuffle Instruction
1) Unified Memory
2) Native SMem atomics
3) Dynamic Parallelism
Hardware support for page faults
Architecture Features
PHJ : Partitioned Hash Join
SMem : Shared Memory
GMem : Global Memory

Fig. 1: Timeline of GPU hash join implementations and GPU architecture features.

In the CUDA programming model, a program executed by the GPU is known as a kernel. A kernel is executed as a grid of thread blocks which can further be broken down into individual threads. Each thread block is assigned to a single SM and the CUDA cores inside each SM executes the threads in a SIMD fashion in groups of 32 threads (known as *warp*). Also, the threads within the same thread block can share the data stored in the GPU shared memory.

B. Emerging GPU Architecture Features

Over the last decade, NVIDIA has introduced the following major architecture features relevant to GPU hash join implementations.

Inter-Thread Communication. NVIDIA has introduced native shared memory atomic operations and shuffle operation for more efficient inter-thread communication. The atomic operations allow the threads within a thread block to modify shared data structures without conflicts; while the shuffle operation which has significantly lower overhead than atomic instructions allow the threads within the same warp to access each other's local memory (registers). Shuffle and shared memory atomic operations make it possible for threads to collaboratively build the histogram or the hash table.

GPU Resource Utilization. To improve the resource utilization, NVIDIA introduced support for multiple CUDA Streams and Dynamic Parallelism. Support for multiple CUDA streams make it possible to simultaneously assign operations to both execution and DMA engines on GPUs. Dynamic parallelism allow kernels to launch their own child kernels, thus making it possible to dynamically alter the number of threads allocated to each partition when joining skewed input relations.

Efficient Data Movement. The unified memory feature allow the GPU driver to move the data between CPU and GPU without explicit memory copy instructions. Further, hardware support for page faults allow more efficient and fine grained movement of data across the PCIe bus when taking advantage of unified memory. The page fault system manages a global page table that is updated every time a data entry is moved from CPU to GPU or vice versa. Further, since the page table needs to be locked for each update, the GPU driver groups together requests from multiple threads into a single page table update.

III. GPU Hash Join Over the Decade

Hash join implementations for CPUs have been studied exhaustively in literature [10]–[12]. In fact, in 2016 Schuh et. al. [13] performed a detailed study of thirteen different join implementations for CPUs. However, this study is specific to CPUs and does not look into GPU hardware and associated implementations. The hash join implementation proposed by He et. al. [1] in 2008 does not take advantage of modern GPU architecture features like atomic operations, unified memory or multiple CUDA streams. In the same year, NVIDIA GPUs started widely supporting atomic operations in global memory. Following this in 2012, Dan et. al. [14] presented a simple non-partitioned hash join implementation which built hash tables using global memory atomic operations. In the same year, Pirk et. al. [15] tried to accelerate foreign-key joins by executing random table lookups on the GPU VRAM.

In 2012, Gregg et. al. [16] demonstrated the impact of PCIe data transfer overhead on the overall performance of GPU applications. At this point, vendors like NVIDIA have been trying to address concerns regarding movement of data and had started supporting features like universal virtual addressing (2010) which allowed GPU kernels to access data directly from the CPU main memory. In 2012, Kaldewey et. al. [6] designed a system that takes advantage of UVA and lock-update-release based atomic instructions (supported since 2010) to share the histogram among multiple threads. Multiple studies on databases in 2013 [2], [8], [17], [18] used non-partitioned hash join implementations. But these implementations failed to outperform the hardware conscious partitioned hash join implementations.

In 2015, Rui et. al. [3] revisited the original hash join implementation and proposed improvements like the use of shuffle instructions (released in 2012) to speedup the prefix sum operation. In 2017, Rui et. al. [4] proposed a hash join implementation that overlaps the data transfer of the second input relation with the partitioning of the first input relation (using CUDA Streams), effectively hiding the cost of this data transfer. The implementation also proposed the use of the Dynamic Parallelism feature released in 2014 to join skewed input relations. Finally in 2019, Sioulas et. al. [9] proposed an implementation that partitions the data using bucket chains, thus avoiding the need to build histograms. This helped the implementation to more efficiently pipeline computation and data transfer compared to previous studies. The same study also presented an implementation that makes use of unified memory (2014) and support for hardware page faults (2016) to move data between CPU and GPU.

IV. Implementation Details

In this section, we first introduce the representative hash join implementations used in this study. We then present the techniques to efficiently handle input relations with high skew or high match rate.

A. Representative Implementations

To conduct a comprehensive study of hash join operation on GPUs, we choose seven different implementations (Table I). In the remainder of this section, we detail the internal working of these implementations with an emphasis on their use of new GPU architecture features.

1) NOP: This is the traditional non-partitioned hash join implementation that adopts a hardware oblivious two-stage design consisting of a build stage and a probe phase. The build phase generates a hash table using global memory atomic operations and the probe phase then probes the hash table using the second input relation. However, this implementation fails to take advantage of GPUs L2 cache and shared memory when the size of the hash table is larger than the L2 cache or shared memory and is hence inefficient for large data sizes.

Join	Paper	Description
NOP	[14]	Non partitioned hash join.
PTH	[1]	Basic parallel radix hash join that partitions the data by building a per-thread histogram.
WSH	This	Same as PTH except the histogram is shared within the same warp using the shuffle instruction.
BSH	[4]	Same as PTH except the histogram is shared within the same thread block using atomic instructions.
PRBC	[9]	Parallel radix hash join that partitions the data using chains of buckets instead of building a histogram.
UMJ	[9]	Same as PRBC except using the unified memory feature to move the data between the CPU and GPU.
UMJ-PF	This	Same as UMJ except using prefetching while accessing input data over PCIe.

TABLE I: Representative hash join implementations.

2) PTH, WSH & BSH: To address the inefficiencies of NOP, implementations like PTH, WSH and BSH recursively partitions both input relations by building histograms in the GPU shared memory. This helps generate small co-partition pairs that fit within the shared memory for the final join operation. In PTH, each individual thread maintains its own histogram data; while WSH and BSH are capable of sharing the histogram among multiple threads (up to 32 and 1024 threads respectively) using shuffle and atomic instructions. To understand the benefit of this sharing, we need to look at how it affects other aspects of the hash join operation, like the occupancy rate (hardware utilization).

Since the shared memory requirements of the threads executing on an SM cannot exceed the shared memory available in hardware, the maximum number of partitions that can be generated in a single pass (P_{max}) by PTH, WSH and BSH is given by Equation 1. Here S_{max} is the maximum amount of shared memory available to the threads allocated to the same SM in bytes, $\#T_{act}$ is the number of threads executing simultaneously per SM, $\#T_H$ is the number of threads sharing the same histogram, $\widehat{H}_{sz}$ is the unit size of a histogram entry in bytes. The value of $\#T_H$ is 1, 32 and 1024 for PTH, WSH and BSH implementations respectively. Further, Equation 1 clearly shows that, $\#T_H$ is inversely related to P_{max} and the level of parallelism ($\#T_{act}$) i.e implementations with lower levels of shared histogram access will have lower occupancy rates for the same number of partitions. This means that PTH, WSH and BSH chooses different levels of trade off between the cost of histogram updates and occupancy rates.

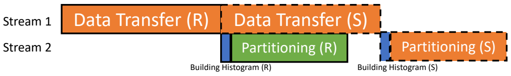

Fig. 2: Execution timeline of BSH.

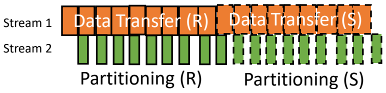

Fig. 3: Execution timeline of PRBC.

$$P_{max} = \frac{S_{max} * \#T_H}{\widehat{H}_{sz} * \#T_{act}} \tag{1}$$

Once both relations are partitioned, the co-partitions are joined using nested loop join. Further, to take advantage of separate DMA and execution engines, WSH and BSH simultaneously executes PCIe data transfer and kernels (using separate CUDA streams). Specifically, both these implementations overlap the partitioning of the first input relation with data transfer of the second input relation as shown in Figure 2.

3) PRBC: The partitioning stage in PRBC does not require the generation of a histogram. It instead makes use of bucket chaining, where empty buckets are allocated to partitions as need. Hence, PRBC is able to more efficiently overlap computation and data transfer as shown in the timeline in Figure 3. This is because, the partition kernel in PRBC offers sufficient computation to overlap the partitioning of each relation with the input data transfer of the same relation, which is not the case for the histogram build operation in BSH or WSH. Finally, PRBC joins each pair of co-partitions using nested loop join or NOP.

4) UMJ & UMJ-PF: All previous implementations make use of explicit memory copy instructions to move the input/output data between CPU and GPU. UMJ and UMJ-PF on the other hand delegates this responsibility to the GPU driver using the unified memory feature. In UMJ, when a thread tries to access data which has not been moved to the GPU, the driver migrates the necessary page to the GPU using the page fault mechanism. Finally, in addition to using the unified memory feature, UMJ-PF enables prefetching of data (using *cudaMemPrefetchAsync* instruction) so that the data is available on the GPU before it is actually required by the kernel threads, thus avoiding the need for page table updates during kernel execution (updates happen during prefetching).

B. Handling Data Skew & Match Rate

In this section, we propose simple techniques to more efficiently join input relations with high skew and high match rate. Note that, techniques proposed in this section can be used to improve the performance all of implementations in Table I. However for the sake of simplicity, we only evaluate these techniques using the BSH implementation (Section V).

1) Data skew: Skewed input relations can lead to severe workload imbalance as data will be unevenly distributed across partitions. Existing implementations either dynamically launch more threads when skewed partitions are detected (BSH) or breakdown partitions into small buckets and distribute the buckets among thread blocks in a round robin fashion (PRBC). However, launching additional kernels at runtime has significant overhead and breaking down partitions among multiple thread blocks is inefficient for uniform data sets [9].

Hence, we adopt a lightweight approach that dynamically re-distributes the thread blocks among the partitions based on their size. For this, we analyze the histogram or bucket headers and then generate a mapping between the thread blocks and the partitions. Since generating the mapping does not require any complex operations, the time taken for this process is often less than 0.5% of the total execution time.

2) Match Rate: Relations with high match rate can cause output buffer overflows when joining co-partitions. Existing solution is to use an additional pass to accurately predict the number of output tuples that will be generated by each pair of co-partitions. This information is then used to group the partitions such that the output generated by each group can fit within the output buffer. However, this additional pass adds unnecessary overhead to the join operation [4]. Further, simply grouping together partitions by estimating worst case output size (from partition size) can lead to the generation of very small groups which cannot efficiently use all GPU cores.

Hence, we adopt an approach that generates partition groups by estimating the number of output tuples than will be generated by each pair of co-partitions. The estimation is based on the size of the co-partitions, the overall input size and the error in previous estimations. This helps achieve higher occupancy rates without encountering output buffer overflows. Finally, we minimize the overhead of recovering from an incorrect estimation by allowing the thread blocks to keep track of available empty slots in the output buffer and the rate at which the buffer is getting filled. This information is then used to detect future overflows and terminate execution.

V. Experimental Evaluation

A. Experimental Setup

Hardware. We use the P100 GPU from NVIDIA for our experiments. The GPU is connected to the CPU via x16 PCIe 3.0 interface. We use CUDA 9.0 and gcc 4.8.5 with full compiler optimization (-O3) to compile the code and use *NVProfile* tool to collect performance metrics.

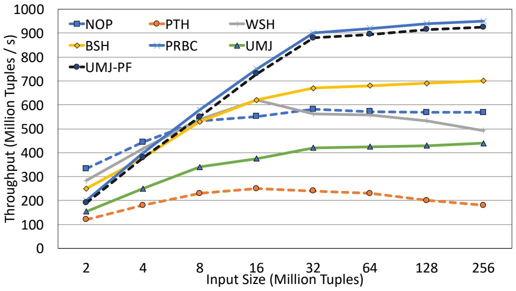

Fig. 4: Performance comparison of all GPU hash join implementation using P100 GPU.

Input Data. For our experiments, we use synthetic data sets following previous studies [3]–[5], [9], [13]. The data set consists of two relations R and S with a tuple size of 8 bytes (4 byte key and 4 byte id). The key values are generated as sequential integers and then randomly shuffled. We further set $|R| = |S|$ for all experiments and vary the total input size (|R| + |S|) from 2M to 256M tuples.

Experimental Outline. In Section V-B, we present the overall performance comparison of the seven representative GPU hash join implementations. Then, in Section V-C we evaluate the impact of using different GPU architecture features. Finally, we evaluate the efficiency of the techniques proposed to handle data sets with high skew and high match rate in Section V-D. Note that, following previous studies [6], [9], [13], we define throughput as the number of input tuples processed per unit time.

B. Overall Comparison

To get a comprehensive understanding of the performance of all seven hash join implementations (Table I), we present their throughput for data sizes from 2M to 256M in Figure 4. We now make the following observations from the results in Figure 4.

First, among the histogram based implementations (BSH, WSH and PTH), WSH achieves the best performance for small data sizes; while BSH outperforms both PTH and WSH by up to 3.9x for large data sizes (Section V-C1). Second, the PRBC implementation that adopts a bucket chaining scheme, outperforms the best histogram based implementation (BSH) by up to 1.36x and achieves over 5.3x performance improvement over the 2008 implementation (Section V-C2). Third, the use of unified memory for data movement (UMJ) leads to significant performance degradation. However, using a combination of unified memory and prefetching (UMJ-PF) helps to achieve performance similar to the PRBC implementation that makes use of explicit memory copy instructions (Section V-C3). We dive deeper into the reasons behind the above observations in the next section.

C. Impact of GPU Architecture Features

In this section, we detail the reasons behind the observation made in Section V-B from the perspective of GPU architecture changes. Specifically, we look at GPU architecture features associated with improving inter-thread communication (Atomic and Shuffle Instructions), hardware utilization (CUDA Streams) and efficiency of data movement (Unified Memory and Hardware Page Fault).

1) Atomic and Shuffle Instructions: To understand how the choice of Atomic (BSH), Shuffle (WSH) or simple arithmetic instructions (PTH) for histogram updates lead to significant performance difference, we need to look back at Equation 1. For the P100 GPU S_{max} is 64K and the value of $\widehat{H}_{sz}$ is 4 (integer data) for all three implementations. Based on this, to generate 1024 partitions in a single pass, PTH needs to set the number of active threads per SM ($\#T_{act}$) to 16; while WSH and BSH can achieve $\#T_{act}$ values of 512 and 2048 respectively. Note that, BSH is limited to 2048 threads as it is the maximum $\#T_{act}$ value supported by the P100 GPU. Hence, the PTH implementation fails to efficiently utilize the GPU hardware and achieves significantly worse performance than WSH and BSH, in-spite of its use of lower overhead instructions for histogram updates.

WSH on the other hand is able to achieve reasonable occupancy rates compared to BSH for small data sets. To demonstrate this, we present the occupancy rates achieved by WSH and BSH for data sizes of 2M and 256M in Table II. The results show that, WSH is able to achieve the same occupancy rate as BSH for small data sizes. This combined with the use of lower cost shuffle instruction helps WSH to achieve better performance for such data sets. However, as the input size increases (which requires larger partition counts) the BSH implementation is able to achieve significantly higher occupancy rates than WSH implementation (based on Equation 1).

Data Size	2M	256M
WSH	90%	24.7%
BSH	90%	89%

TABLE II: Occupancy rates of WSH and BSH for 2M and 256M tuples.

2) CUDA Streams: To understand the performance difference between BSH and PRBC we to look at how these implementations take advantage of CUDA Streams. As see in Section IV, both BSH and PRBC overlapps data transfer and computation using multiple CUDA streams. However, the PRBC implementation is able to more efficiently overlap data transfer and computation (Figures 2 and 3). To demonstrate this, we present the throughput achieved by PRBC and BSH when the use of multiple CUDA streams is disabled in Figure 5. The results clearly show that when the use of multiple CUDA streams is disabled, BSH is able to achieve performance close to PRBC.

3) Unified Memory & Hardware Page Fault: The GPU driver groups together page table updates from multiple threads when using the unified memory feature. To minimize the number of unique page table updates and associated overhead. However, it is not possible to group together the requests from all active threads. In fact, the UMJ implementation encounters over 6K unique page table updates for a data size of 256M tuples (collected from NVProfile). This is why the use of the unified memory feature leads to significant performance degradation in UMJ, when compared to PRBC. UMJ-PF on the other hand is able to avoid page table updates during kernel execution by prefetching data. Further, UMJ-PF is able to hide the cost of prefetching by overlapping this operation with execution of unrelated threads. Hence, it is able to achieve performance similar to the PRBC implementation. Note that, the performance of UMJ-PF is still lower than PRBC due to additional work that needs to be done by UMJ-PF to update the page tables during data prefetching (not required when using explicit memory copy instructions).

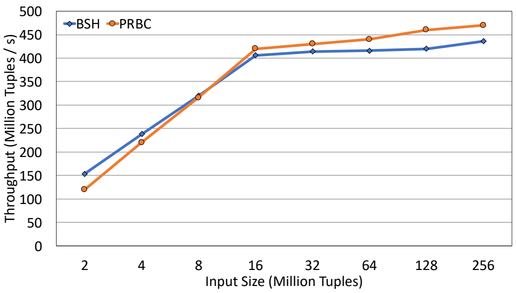

Fig. 5: Performance of BSH & PRBC with CUDA Streams disabled.

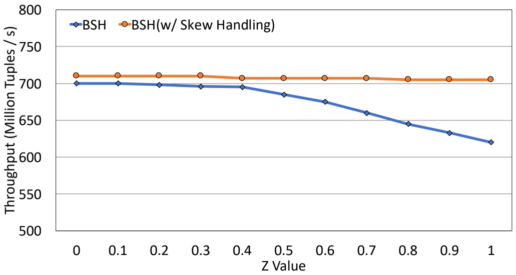

Fig. 6: Performance comparison of BSH & BSH(w/ Skew Handling) under varying skew levels.

D. Handling Data Skew & Match Rate

To demonstrate the benefits of our skew handling technique, we present the throughput achieved by BSH and BSH(w/ Skew Handling) when joining a skewed data set (256M tuples) that follows the zipf distribution (z values from 0 to 1). Note that, BSH addresses data skew by launching additional threads at runtime (using Dynamic Parallelism); while BSH(w/ Skew Handling) dynamically re-distributes threads blocks (Section IV). The results show, that BSH encounters over 12% performance degradation as the skew increases; while BSH(w/ Skew Handling) shows absolutely no performance degradation even for very high z factor values. Note that, BSH(w/ Skew Handling) performs better than BSH even for small z factor values due to the higher overhead of detecting skews in the BSH implementation.

To demonstrate the efficiency of the technique proposed to efficiently join input relations with high match rate, we present the throughput achieved by BSH, BSH(w/ Baseline Estimation) and BSH(w/ Estimation) when the match rate

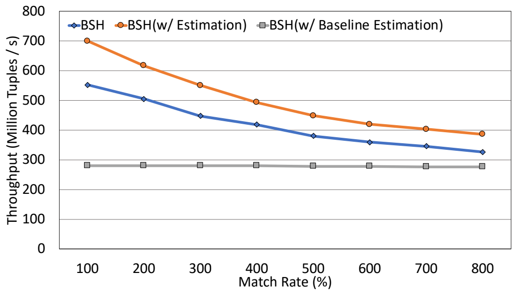

Fig. 7: Performance comparison of BSH, BSH(w/ Baseline Estimation) & BSH(w/ Estimation) under varying match rate.

is increased from 100% to 800% (input size is 256M). As discussed before, BSH groups together partitions using an additional pass; while BSH(w/ Baseline Estimation) and BSH(w/ Estimation) groups together partitions by estimating the possible output size. When compared to BSH(w/ Estimation), BSH(w/ Baseline Estimation) makes the worst case estimation. The results show that, BSH(w/ Estimation) outperforms BSH and BSH(w/ Baseline Estimation) by up to 1.27x and 2.5x respectively. This is because, BSH(w/ Estimation) does not require an additional pass compared to BSH and makes better estimations than BSH(w/ Baseline Estimation), which helps generate larger groups. Note that, the throughput achieved by all the implementations drop significantly as the match rate increases, due to the higher cost of materializing output data.

VI. Findings & Future Opportunities

In this section, we highlight the major experimental findings and present opportunities for future research. The important findings from Section V include:

- Modern GPUs support varying levels of inter-thread communication: from high overhead and less resource intensive to low overhead and high resource intensive. The choice of the right scheme depends on the workload as well as the hardware being used.
- The computational capability of GPUs have improved significantly over the years; while the PCIe bus has shown very minimal throughput improvement. Further, new faster interconnects like NVLink are currently not supported by CPU vendors like Intel and AMD. Hence, the movement of data between CPU and GPU is a major bottleneck and systems designers should favour implementations that can achieve maximum overlap of data transfer and computation.
- By adding data prefetching, unified memory based implementations on modern GPUs can achieve performance similar to implementation using explicit/manual memory copy while also minimizing the complexity of memory management.

Based on the above findings and the results in the Section V, it is clear that there are a number of parameters (e.g. tile size, partition count, thread configuration etc.) that needs to be tuned when taking advantage of certain GPU architecture features. Further, optimal values for these parameters often depend on a number of factors like input size, hardware configuration etc. Hence, a study on determining the optimal values for these parameters could be an interesting future work.

VII. Conclusion

In this paper, we revisit the major GPU hash join implementations in the last decade and detail how they take advantage of different GPU architecture features. Our study finds that, by efficiently taking advantage of GPU architecture features like CUDA Streams and shared memory atomic operations, PRBC outperforms all other existing GPU hash join implementations. The study also sheds light on the impact of different architecture features on the hash join operation and has identified a number of factors guiding the choice of these features. Finally, the techniques proposed in this study help avoid any performance degradation when joining data sets with high skew and achieves up to 1.27x performance improvement when joining data sets with high match rate.

VIII. Acknowledgement

This work is supported by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) in Singapore.

References

[1] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, “Relational joins on graphics processors,” in *SIGMOD*, 2008.

[2] J. He, M. Lu, and B. He, “Revisiting co-processing for hash joins on the coupled cpu-gpu architecture,” *Proc. VLDB Endow.*, 2013.

[3] R. Rui, H. Li, and Y. C. Tu, “Join algorithms on gpus: A revisit after seven years,” in *ICBD*, 2015.

[4] R. Rui and Y.-C. Tu, “Fast equi-join algorithms on gpus: Design and implementation,” in *SSDBM*, 2017.

[5] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima, “Relational joins on gpus: A closer look,” *TPDS*, 2017.

[6] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, “Gpu join processing revisited,” in *DaMoN*, 2012.

[7] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, “Kernel weaver: Automatically fusing database primitives for efficient gpu computation,” in *MICROArch*, 2012.

[8] J. He, S. Zhang, and B. He, “In-cache query co-processing on coupled cpu-gpu architectures,” *Proc. VLDB Endow.*, 2014.

[9] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, “Hardware-conscious Hash-Joins on GPUs,” *ICDE*, 2019.

[10] S. Blanas, Y. Li, and J. M. Patel, “Design and evaluation of main memory hash join algorithms for multi-core cpus,” in *SIGMOD*, 2011.

[11] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu, “Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware,” in *ICDE*, 2013.

[12] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, “Sort vs. hash revisited: Fast join implementation on modern multi-core cpus,” *Proc. VLDB Endow.*, 2009.

[13] S. Schuh, X. Chen, and J. Dittrich, “An experimental comparison of thirteen relational equi-joins in main memory,” in *SIGMOD*, 2016.

[14] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, “Building an efficient hash table on the gpu,” in *GEMS*, 2011.

[15] H. Pirk, S. Manegold, and M. Kersten, “Accelerating foreign-key joins using asymmetric memory channels,” in *ADMS*, 2011.

[16] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate cpu vs. gpu performance without the answer,” in *ISPASS*, 2011.

[17] Y. Yuan, R. Lee, and X. Zhang, “The yin and yang of processing data warehousing queries on gpu devices,” *Proc. VLDB Endow.*, 2013.

[18] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, “Hardware-oblivious parallelism for in-memory column-stores,” *Proc. VLDB Endow.*, 2013.