Join Algorithms on GPUs: A Revisit After Seven Years

Ran Rui, Hao Li, and Yi-Cheng Tu
Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave., ENB 118
Tampa, Florida, USA
{*ranrui,haoli1*}*@mail.usf.edu, ytu@cse.usf.edu*

***Abstract*—Implementing database operations on parallel platforms has gain a lot of momentum in the past decade. A number of studies have shown the potential of using GPUs to speed up database operations. In this paper, we present empirical evaluations of a state-of-the-art work published in SIGMOD'08 on GPU-based join processing. In particular, this work presents four major join algorithms and a number of join-related primitives on GPUs. 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. Our experiments on today's mainstream GPU and CPU hardware show that the GPU join program achieves up to 20X speedup in end-to-end running time over a highly-optimized CPU version. This is significantly better than the 7X performance gap reported in the original paper. 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. By applying such optimizations, extra performance improvement of 30-52% is observed in various components of the GPU program. Finally, we evaluate the same program from a few other perspectives including energy efficiency, floating-point performance, and program development considerations to further reveal the advantages and limitations of using GPUs for database operations. In summary, we find that today's GPUs are significantly faster in floating point operations, can process more on-board data, and achieve higher energy efficiency than modern CPUs.**

***Keywords*-DBMS; GPU; GPGPU; CUDA; join operation;**

I. Introduction

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) [1] and Open Computing Language (OpenCL) [6], 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 [27, 16, 17, 25, 20, 11, 18, 26, 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 [17]: 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 paper, we report the results of a comprehensive set of experiments running the program developed by He *et al.*. However, our study serves more significant purposes than simply verifying the findings of [17]. Instead, we aim at drawing an up-to-date and panoramic image of GPGPU as a means for processing join operators.

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 I shows the specifications of several Nvidia GPUs, including the 8800 GTX that is used as the testbed in [17]), and the GTX Titan plus GTX 980 that we use in our study.[1] 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 [17] 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

[1]Information is mainly extracted from the Intel and Nvidia corporate websites, with other information obtained from www.techpowerup.com

Table I
SPECIFICATIONS OF HARDWARE MENTIONED IN THIS PAPER

Device	CPU			GPU		
	Xeon E5-2640 V2	Core i7 3930K	Core 2 Quad 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	14×192	16×128	8×32
Cache Size	L1: $512KB$ L2: 2MB	L1: $384KB$ L2: 1.5MB	L1: $256KB$ L2: $8MB$	L1: $64KB \times 14$ L2: 1536KB	L1: $96KB \times 16$ L2: 2MB	L1: $16KB \times 8$
RAM	DDR3 Triple Channel		DDR2 Dual Channel	GDDR5 6GB 384 bit	GDDR5 4GB 256 bit	GDDR3 768MB 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

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 [17] 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.

The remainder of this paper is organized as follows: We briefly review related work in Section II; The experimental setup is described in Section III; We report performance of the original join code used in [17] in both CPUs and GPUs in Section IV; We present design and evaluation of optimized join algorithms based on new GPGPU features in Section V; We continue to evaluate the GPU join algorithms from other perspectives in Section VI, summarize our findings in Section VII, and conclude this paper by Section VIII.

II. 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 – more than 9% of the Top500 supercomputers in the world has deployed GPUs in their architecture [8]. 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.* [23] 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.* [12] extended that proposal to a practical scenario by integrating GPU-assisted spatial operations into a commercial DBMS. Govindaraju *et al.* [14] 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 [13]. 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 [9], there has been unprecedented effort devoted to the GPGPU paradigm, especially after the release of advanced GPU computing models such as CUDA [1] and OpenCL [6]. The same trend has also affected the database community. He *et al.* [15] 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. [17] 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 [16] of [17], 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 [17] for comparison and dicsussions in this paper. In [20], 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.* [11] 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 [16, 19].

III. 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 [7] for efficient data transfer between the host and the GPU.

We obtain the entire code package introduced in [17] 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 paper 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 I. 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 [4]. 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 paper. Interested readers can refer to a longer version of this paper [22] 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 [17]. As in [17], 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 [10] 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 run-time power readings under the assumption that power does not change within the sampling window.

IV. 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.

A. 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 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 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

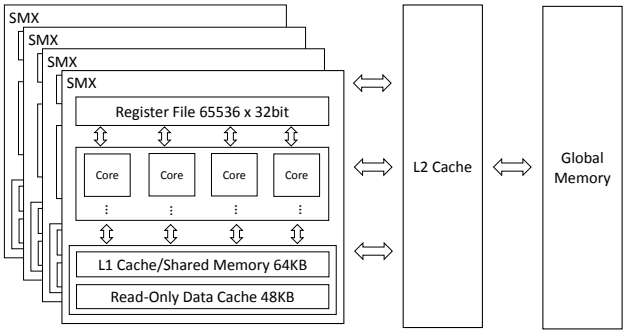

Figure 1. Memory hierarchy in Kepler architecture

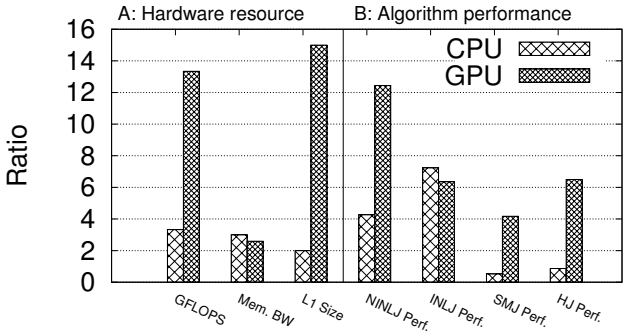

Figure 2. Relative capacity of hardware resources and join performance between new and old CPUs/GPUs

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.

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.

B. 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.[2] 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 I). Therefore, the two speedup values shown in each row of Table II 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.[3]

In most cases, the recorded speedup beats the corresponding value reported in [17] (shown in the *Baseline* column of Table II). The largest difference between the recorded speedup and baseline comes from the SMJ algorithm: even the smallest speedup value is a few times higher than the 2.4X reported in [17]. 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 [17] 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 VI-C. 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.

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

[2]In all cases, the variance of the four runs is very small, indicating stable performance of both CPU and GPU code.

[3]Not exactly true for SMJ, but close enough as the performance of GTX980 is almost the same as Titan.

Table II
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	Baseline
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	–

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 2B, along with the relative specifications between the same set of hardware shown in Figure 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 [17]. 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 II.

Relating the information in Figure 2B to the hardware information in Figure 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.[4]

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 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 [17]. 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% 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 [17]. 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 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.

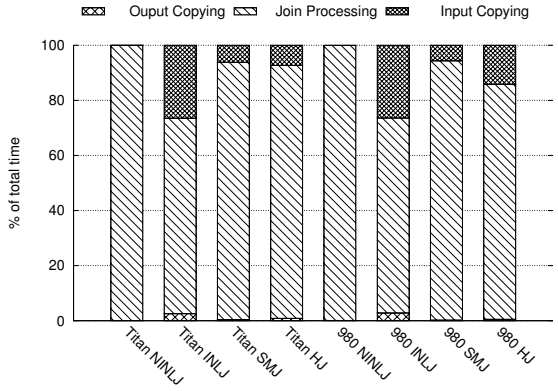

Figure 3. Time spent on data transmission and join processing

V. 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

[4]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.

change of the code structure. A systematic re-design of GPU join algorithms is beyond the scope of this paper.

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 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 [2] 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.

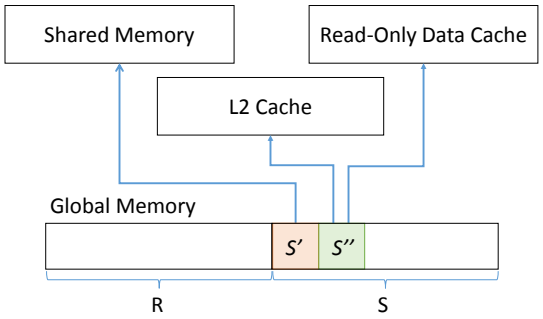

Figure 4. Data movement in the modified NINLJ algorithm. S' and S" are two blocks of table S

A. 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 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

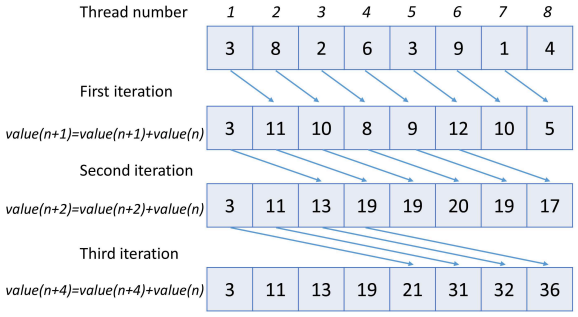

Figure 5. Data access pattern using shuffle instructions

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 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.

B. Performance Evaluation

Table III 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 II), **the average Titan-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 IV 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 III
NINLJ PERFORMANCE ON GTX TITAN UNDER READ-ONLY AND L2 CACHE OPTIMIZATIONS

Data	Running time (sec)			Speedup	
Size	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 IV
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

VI. 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 [17] but without much quantitative results.

A. 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 II, the average power consumption are shown in Table V. 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.

Table V
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	–

As to the total active energy consumption, it is obvious that in most cases the i7-3930K consumes the most energy (Figure 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 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 [22].

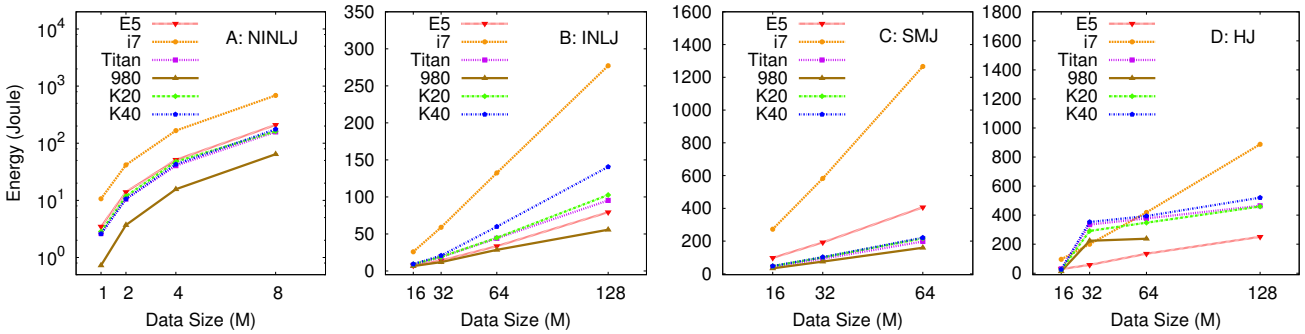

Figure 6. Energy consumption of different CPUs/GPUs in processing four join algorithms

B. 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 [3]. 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 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 II). 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 II. 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.

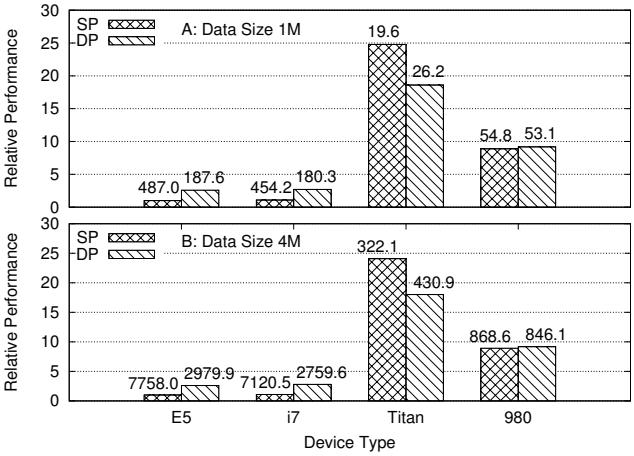

Figure 7. Relative performance of NINLJ with SP/DP keys in different CPUs/GPUs under two table sizes

C. 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 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.

VII. 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 [17].

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 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 [21] 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 I). 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 [5].

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 V) 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.

Response to concerns shown in [17]: 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 VI-A). 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 VI-B).

VIII. Conclusions and Future Work

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.

Work in this topic can be extended along two directions. First, we could continue the evaluation of the join algorithms on emerging GPU hardware and software. For example, the coming Pascal architecture promises memory bandwidth and size that are a few times higher than those in Kepler. It would be interesting to have close observations on the race between such GPUs and other multi/many-core processors in processing database operators. Second, design of join algorithms optimized towards new architectural features, as suggested by our work, deserves more attention. Features such as fast links between host and GPU can be a game-changer but also calls for re-hauling of the algorithm design. In today's big data applications, joins can be performed on tables that are too big to fit in the global memory of one GPU device. Therefore, there are urgent needs to consider distributed versions of GPU joins.

Acknowledgements: The project described was supported by an award (CAREER, IIS-1253980) from US National Science Foundation (NSF). A part of this work was support by NSF grant IIS-1117699.

REFERENCES

[1] CUDA parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html.

[2] Maxwell: The Most Advanced CUDA GPU Ever Made. http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made.

[3] Nvidia Kepler GK110 Architecture Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[4] NVIDIA Tesla. http://www.nvidia.com/object/tesla-workstations.html.

[5] Nvidia updates GPU Roadmap; Announces Pascal. http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal.

[6] OpenCL. https://www.khronos.org/opencl.

[7] Peripheral Component Interconnect Express. http://en.wikipedia.org/wiki/PCI_Express.

[8] TOP 500 List. http://www.top500.org/lists/2014/06.

[9] Unified Shader Model. http://en.wikipedia.org/wiki/Unified_shader_model.

[10] Watts Up Power Meters. https://www.wattsupmeters.com/secure/products.php?pn=0.

[11] P. Bakkum and K. 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.

[12] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *Procs. 13th Intl. Conf. on Very Large Data Bases*, VLDB '04, pages 1021–1032, 2004.

[13] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 325–336, 2006.

[14] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 215–226, 2004.

[15] B. He, N. K. Govindaraju, Q. Luo, and B. 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.

[16] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[17] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 511–524, 2008.

[18] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Procs. VLDB Endowment*, 4(5):314–325, Feb. 2011.

[19] J. He, M. Lu, and B. He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endowment*, 6(10):889–900, Aug. 2013.

[20] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *Procs. 8th International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62, 2012.

[21] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, Dec. 2000.

[22] R. Rui, H. Li, and Y.-C. Tu. Performance Analysis of Join Algorithms on GPUs. Technical Report CSE/14-016, Department of Computer Science and Engineering, University of South Florida, 2014.

[23] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 455–466, 2003.

[24] Y.-C. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler. Data Management Systems on GPUs: Promises and Challenges. In *Procs. 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 33:1–33:4, 2013.

[25] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Procs. IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, 2014.

[26] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endowment*, 6(10):817–828, Aug. 2013.

[27] Y. 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.