Performance Analysis of Join Algorithms on GPUs

Ran Rui
Department of Computer Science and Engineering
University of South Florida
ranrui@mail.usf.edu

Hao Li
Department of Computer Science and Engineering
University of South Florida
haoli1@mail.usf.edu

Yi-Cheng Tu
Department of Computer Science and Engineering
University of South Florida
ytu@cse.usf.edu

ABSTRACT

Implementing database operations on parallel platforms has gain a lot of momentum in the past decade, due to the increasing popularity of many-core processors. 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, such work provides four major join algorithms and a number of join-related primitives. 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 over a highly-optimized CPU version. This is significantly better than the 7X performance gap reported in the original paper. We also modify the GPU programs to 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 achieves higher energy efficiency than modern CPUs. The availability of new tools and models have made program development and optimization on GPUs much easier than before.

1. INTRODUCTION

The design of modern CPUs has experienced a reform from single-core to multi-core architecture. The single-core designs reached a bottleneck where instruction-level parallelism is too exhausted to continue providing performance improvements. On the other hand, the strategy of further increasing clock frequency hits an "energy wall" in that the high power leakage of the chips becomes intolerable [11]. Although multi-core CPUs have become more power efficient than ever before, the growth of their performance can hardly catch up with Moore's Law any more. As a result, Many-core architectures such as Graphics Processing Units (GPU) have become a popular choice of high-performance computing (HPC) platform. GPUs once were dedicated hardware for computer graphic rendering tasks such as gaming, 3D design, and video processing. Its architecture is uniquely designed towards rendering tens of video frames that each contains millions of pixels within a second. 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. Various studies have shown that GPU is promising for accelerating data-independent parallel computing, especially in applications such as fluid dynamics, molecular simulation and data mining [28, 30, 25, 24, 37, 26, 34]. Such work reported performance boosts (over CPUs) ranging from one to three orders of magnitude, depending on the level of parallelism that can be exploited in the algorithms.

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 [38, 19, 20, 33, 23, 12, 21, 35, 32]. In relational DBMSs, the most time-consuming operation is join, which is accomplished by either exhaustively scanning the whole table or accessing accessory data structures (e.g., indexes) to reduce search space. In 2008, He *et al.* published their work in the design and implementation of four major join algorithms on GPUs [20]: 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 [20]. 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 1 shows the specifications of several Nvidia GPUs, including the 8800 GTX that is used as the testbed in [20]), and the GTX Titan plus GTX 980 that we use in our study. 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 [20] 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 the GPUs achieve up to a 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 modify 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 program development considerations. Those are done in response to relevant discussions presented in [20] and further revealed 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 and models has made program development and optimization on GPUs much easier than before.

Roadmap: The remainder of this paper is organized as follows: We briefly review related work in Section 2; The experimental setup is described in Section 3; We report performance of the original join code used in [20] in both CPUs and GPUs in Section 4; We present design and evaluation of optimized join algorithms based on new GPGPU features in Section 5; We continue to evaluate the GPU join algorithms from other perspectives in Section 6, summarize our findings in Section 7, and conclude this paper by Section 8.

2. RELATED WORK

In the database domain, multi-core CPUs and various many-core devices have been used for increasing the performance of various database operations. The key ideas of algorithm design and optimization in this field include minimizing overhead of synchronization, reducing frequency of inter-core communications, optimizing memory access pattern for maximum bandwidth utilization and decreasing cache miss rate. In [14], Blanas *et al.* studied a number of main memory hash join algorithms that took advantages of multi-core CPUs. The authors implemented four flavors of hash joins and demonstrated that the one with a single shared hash table performed better than all other alternatives in most cases. Zagha *et al.* [36] implemented radix sort on Cray multiprocessors that laid out the foundation for radix hash join and provided important theoretical guidance for parallel implementations. Moreover, in dealing with the speed gap between processor and memory, Manegold *et al.* [29] evaluated the impacts of memory cost of database operations and showed how memory optimizations address such issues.

GPGPU has become very popular high-performance computing technique in the last few years. The SIMD architecture of GPU provides tremendous amounts 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]. In computational sciences, GPUs are widely embraced by researchers from various fields to accelerate their applications [28, 30, 25, 24, 37, 26, 34]. It is inevitable that the database community delves into integrating GPGPU into the implementation of DBMSs. 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.* [31] 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.* [13] extended that proposal to a practical scenario by integrating GPU-assisted spatial operations into a commercial DBMS. Govindaraju *et al.* [17] 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 [16]. However, the studies mentioned above were all based on very old GPU architectures, which were not optimized for general-purpose computation. They also had to rely on graphic APIs such as OpenGL and DirectX that 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.* [18] 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.* [20] developed a comprehensive package of GPU-based database algorithms including a series

Table 1: Specifications of hardware mentioned in this paper. Information is mainly extracted from the Intel and Nvidia corporate websites, with other information obtained from www.techpowerup.com

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

of primitives and four join algorithms developed on top of those primitives. The algorithms took advantage of coalesced memory access and shared memory to reduce memory latency and initiated sufficient number of threads to hide memory stalls. 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 [19] of [20], 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 [20] for comparison and discussions in this paper. In [23], 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.* [12] 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 [19, 22].

3. 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 [20] 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 focus on 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 Figure 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 three other GPU products: the GTX 770, Tesla K20, and K40 [4], all belonging to the Kepler architecture. Note that these devices are way more expensive than (yet with only comparable performance as) the Titan therefore we just briefly discuss their performance in Appendix A.

Our workstation is a dual-boot platform that runs the following systems: (1) Windows 7 (SP1) with Visual Studio 2010 as the program development environment; and (2) Ubuntu 13.10 with Linux kernel version 3.11.0, and GCC/G++ version 4.8.1 for compiling the CPU code. For GPU computing, we use CUDA 6.0 to compile the GTX Titan code and CUDA 6.5 for the GTX980. The code was originally developed and tested using Visual Studio in Windows. To ensure fair comparisons, unless specified otherwise, we report GPU results under the Windows system throughout the paper. For the convenience of our future work, we ported the GPU code to Linux without much modification. We tested the GPU code under both Linux and Windows and found that the change of operating system has minor effects on the performance of GPU code. However, some parts of CPU code were implemented by using Windows-oriented libraries. Therefore, we conduct the CPU tests strictly in Windows. The code was compiled and tested with the best configura-

tion and parameters discussed in the previous work [20]. As in [20], 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. In order to test the double precision computing power of CPUs and GPUs, we transform the key values into double precision floating point numbers by modifying the random number generator.

Performance measurement is done by the build-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 runtime power readings under the assumption that power does not change within the sampling window.

4. 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.

4.1 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 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 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.

Comparing with previous architectures such as that of the 8800 GTX, the Kepler architecture [3] carries much more cores as well as larger shared memory and cache. This is the key source of GPU's increasing computing power. Obviously, more cores can provide more resources for supporting larger number of threads concurrently. Also there are subtle hardware improvements, such as faster atomic operation, better double-precision performance, dual instruction

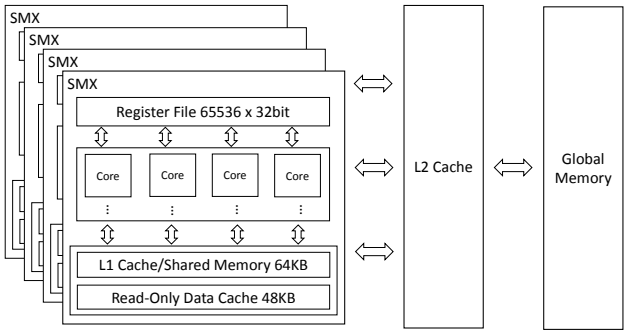

Figure 1: Memory hierarchy in Kepler architecture

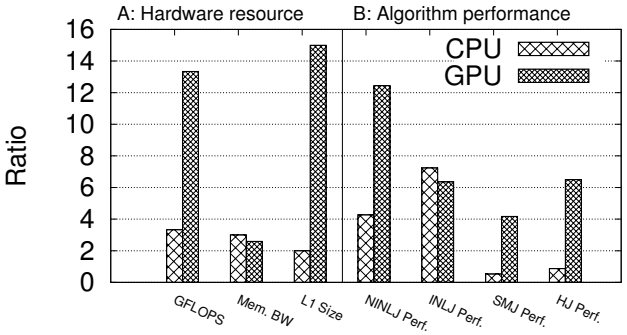

Figure 2: Relative capacity of hardware resources and join performance between new and old CPUs/GPUs

dispatch unit, all of which make the GPU hardware more capable in general-purpose computing. Note that the GTX 980 belongs to the latest Maxwell architecture [2], which is not much different from Kepler except for larger shared memory and higher compute efficiency.

As a relatively new type of hardware, the GPU's computing power has grown in a manner close to the Moore's Law. On the other hand, even with multi-core technology, it becomes more difficult for the CPUs to follow the same trend. This can be seen from Figure 2A, in which we plot the relative capacity of different on-board hardware resources of a new GPU (GTX 980) to an old one (8800 GTX), as well as such of a new CPU (Xeon E5-2640) to its predecessor (Core 2 Q6600). In particular, the GPU in-core computing power becomes 13.4X in a roughly 7-year period while that of the CPU reaches only 3.3X. For the size of L1 cache, the number is 15X for the GPU and only 2X for CPU. The growth of memory bandwidth on the GPUs (2.6X) is slightly smaller than that of the CPUs (3X).

In accordance with the growing computing power, the deployment of PCI-E 3.0 standard makes data copying between main memory and GPU's global memory much faster than before. The time for such data transfer is regarded as an extra cost in GPGPU computing. The top part of Figure 3 shows the data transfer rate achieved in our experiments run under different data block sizes on the new GPUs. The achieved bandwidth is around 10GB/s, in comparison to the theoretical value of 15.8GB/s. On the other hand, this is significantly higher than the 3.1GB/s peak bandwidth on the 8800 GTX reported by [20]. The 8800 GTX connects to the host machine via PCI-E 1.0, with a theoretical bandwidth of 4GB/s. We also verified the global memory bandwidth of

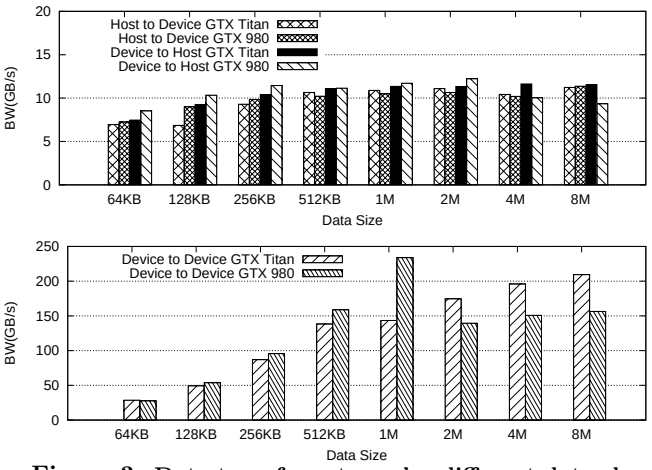

Figure 3: Data transfer rate under different data chunk sizes

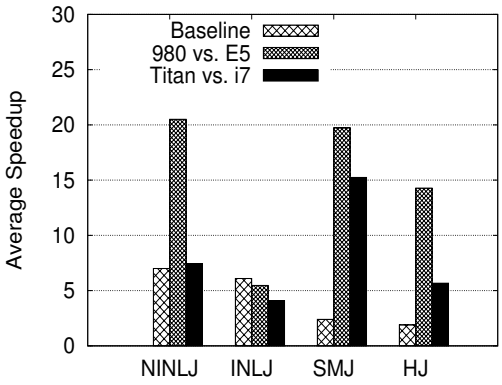

Figure 4: Average GPU-to-CPU speedup

the new GPUs (bottom part of Figure 3): a data transfer rate between 150GB/s and 200GB/s is observed. We notice that in order to achieve optimal bandwidth utilization, moving a large chunk at a time is necessary.

Apart from performance, 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.

4.2 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.[1] 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 1). Therefore, the two speedup values shown in each row of Table 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.[2]

In most cases, the recorded speedup beats the corresponding value reported in [20] (shown in the *Baseline* column of Table 2). 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 [20]. 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 [20] 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 6.3. Note that the above problem cannot be fixed by simply changing the partition size – the performance "valley" at particular input sizes always exists. These are represented by the missing values in Table 2. The average speedup over all data sizes are plotted in Figure 4. 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.

Performance analysis via profiling: Nvidia provides intensive performance profiling via the *Visual Profiler* as part of its CUDA Toolkit. Basically, the profiler reports the quantity of about two dozen hardware counters during the lifetime of a CUDA kernel and performs a detailed qualitative analysis on performance bottlenecks. Such information is invaluable in understanding the behaviour, and in turn, further optimizing the GPU code. For our work, we can use such information to (partially) interpret the GPU-to-CPU speedup data shown above. In Table 3, we list the utilization of the most relevant hardware resources recorded by the Visual Profiler in running the main kernels of all four

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

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

Table 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	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	–

GPU join algorithms. The missing values represent zero or negligible utilization that is not reported by the profiler. The kernels are ordered by their weight in terms of relative running time within each algorithm. The impressive performance of NINLJ in GPUs is clearly the result of high utilization of cache, especially the shared memory. On the CPU side, although profiling results are not available, we can see why they cannot rival the GPUs in running NINLJ: the E5-2640, for example, has 512KB of L1 cache that has to be shared by the entire system; but the GTX980 has a total of 1.5MB of L1-level shared memory that is dedicated to cache the table blocks. We also notice that, as data is fed into the arithmetic units in cache speed, the utilization of the later also reaches a high level (84% and 55%). By this, the GPUs enter their comfort zone due to their arithmetic computing power that is tens of times higher than CPUs (Table 1). The situation of SMJ is similar: two kernels enjoy very high utilization of L2 cache and considerable utilization of L1 cache. The other kernel (*bitonic*) is bound by global memory bandwidth, but it does not hurt the performance significantly with its 23% and 11% utilization of L2 and L1 cache. The INLJ is a different story: all three kernels are found to be bound by memory latency, although some cache hits are observed. In particular, the random access manner in searching the tree causes code divergence, which leads to low level of parallelism that is not sufficient to hide the latency. This can be verified by its extremely low utilization of compute units. Another factor that limits the speedup of INLJ is the overhead of transferring input/output (Figure 5). Among all algorithms, such overhead is the biggest in INLJ as the total amount of work for join processing is the smallest – the can be seen from the total running time. For HJ, all kernels use the shared memory and L2 cache to some extent therefore manage to feed the compute unit with some work, especially in *Probe_Write*. That explains why the HJ has better speedup over CPUs than the INLJ.

In summary, cache utilization seems to be the most important factor in harnessing the capabilities of GPUs. Only through the high bandwidth of cache can the large computing capabilities of GPUs be released. Modern GPUs are also well designed towards that direction with a significant amount of L1/L2 cache. The programmability of L1-level shared memory, we believe, is a big plus by giving programmers the opportunity to fully explore data locality. The profiler provides abundant information for us to analyze the program behaviour therefore is a much welcomed tool. On the other hand, we understand a complete performance analysis needs other information, such as hardware utilization of the CPU code. Although profiles of CPU code are not available, the following discussions may derive some insights.

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 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 [20]. 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 2.

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

Table 3: Resource utilization of major GPU join kernels on GTX980

Algorithm	Kernel Name	Running Time	GM B.W.	SM B.W.	L1 Cache B.W.	L2 Cache B.W.	Arithmetic Op. Units
NINLJ	*write*	71.4%	0.03%	45.2%	0.15%	0.3%	84%
	gpuNLJ	28.6%	0.13%	95.2%	0.48%	1.0%	55%
INLJ	*gSearchTree*	63.3%	19.23%	–	20.29%	55.4%	–
	gJoinWithWrite	24.2%	31.46%	–	3.97%	11.6%	–
	gIndexJoin	12.1%	28.44%	–	3.57%	9.6%	–
SMJ	*bitonic*	55.7%	76.81%	–	11.34%	23.0%	5%
	partBitonicSort	32.0%	23.45%	–	43.62%	96.0%	6%
	unitBitonicSort	10.6%	4.79%	–	47.03%	95.6%	0.1%
HJ	*Probe_Write*	23.2%	11.58%	25.6%	2.05%	3.9%	56%
	Reorder3	21.8%	20.65%	2.6%	2.92%	6.8%	4%
	Reorder	21.1%	21.51%	2.7%	2.73%	6.6%	4%
	Histo	10.2%	9.21%	5.9%	1.61%	2.8%	8%

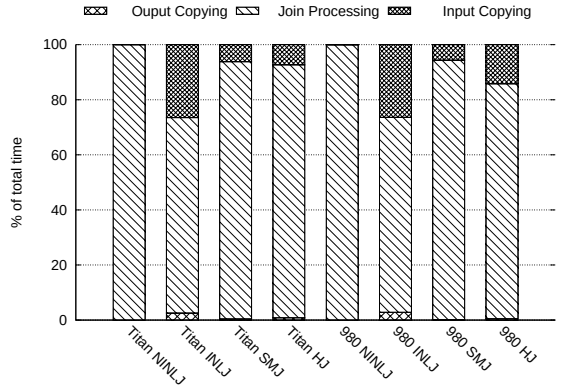

Figure 5: Time spent on data transmission and join processing

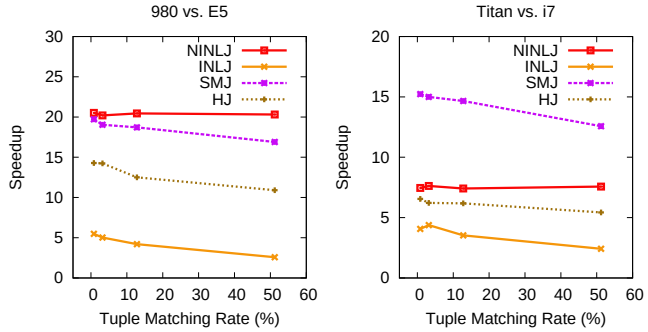

Figure 6: GPU-to-CPU speedup of four join algorithms under different join selectivity

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.[3]

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 5 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 [20]. 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 [20]. 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 according to 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 according to Figure 3.

Effects of join selectivity: Figure 6 shows the GPU-to-CPU speedup of four join algorithms under different selectivity of the join conditions. Lower selectivity results in more matching tuples, thus increases the size of output. According to Figure 6, the four joins have stable running time on both CPU and GPU under smaller tuple matching rates. Such observation is similar to those shown in Figure 13 of [20]. It indicates the data transfer rate between host memory and GPU device memory is sufficient for such workload and therefore has minor effects on the performance of GPU. However, as we increase the tuple matching probability to 50%, we see a decrease of speedup for most joins. This is not surprising as a 50% matching rate will generate a large number of output values thus increase the overhead of transmitting output to the host memory for GPU programs to a great extent (Figure 5). On the other hand, such overhead is still small comparing to the large amount of computations performed in the main body of the NINLJ algorithm – its speedup over CPU does not change even under the 50% tuple matching rate.

Performance of other kernels: We also compare the performance of primitives that serve as building blocks of the four join algorithms, which is shown in Table 4. From the figure we see GPU has 4.7-39.3X speedup over CPU. Compared with [20], most primitives achieve much better

[3]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.

Table 4: Running time (ms) of join-related data primitives

Algorithm	Xeon E5	Core i7	GTX Titan	GTX 980	980-E5 speedup	Titan-i7 speedup	Baseline
map	550	364	15	14	39.3	24.3	27.3
scatter	2274	1815	235	196	11.6	7.7	12.6
gather	4550	3016	104	149	30.5	29.0	9.7
prefix scan	400	348	38	24	16.7	9.2	10.1
split	1056	691	147	169	6.2	4.7	6.5
qsort	4594	3409	284	241	19.1	12.0	2.4

speedup. However, we notice that *scatter* and *split* operations have smaller lower speedup than the baseline. For both operations, the problem is much like the situation in HJ: a kernel is always launched with only 32 threads (a warp) within a thread block. This might not be devastating on earlier GPU device. However, those 32 threads are not able to fully occupy a multiprocessor in newer generation GPUs, making the multiprocessor underutilized.

5. OPTIMIZATION ON NEW GPU ARCHITECTURE

In this section, we demonstrate how features in latest generations of GPUs affect join performance. Note that we focus on mechanisms that can be implemented without a disruptive change of the code structure. A systematic redesign of GPU join algorithms is beyond the scope of this paper.

The GPUs have a carefully designed memory system that allows for maximum throughput feeding the large number of cores and minimizing memory stalls. The global memory in modern GPUs is based on GDDR5 technology with a bandwidth up to 300GB/s. However, if the memory is not accessed in a coalesced manner, the high latency (i.e., a few hundred cycles) of GDDR5 can easily make it the performance bottleneck in various applications. If the data to be read has some locality, the cost of accessing global memory can be lowered by utilizing the cache. Starting from Fermi architecture, Nvidia GPUs are equipped with L1/L2 cache with comparable performance as their CPU counterparts. The L1/L2 cache cannot be controlled by programmers but provides extra performance when locality exists in global memory access. Part of the L1 cache is set aside as a programmable section named the *shared memory*. Due to its high performance and programmability, the shared memory has been widely used for optimizing GPU applications. The original join code we tested is of no exception.

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.

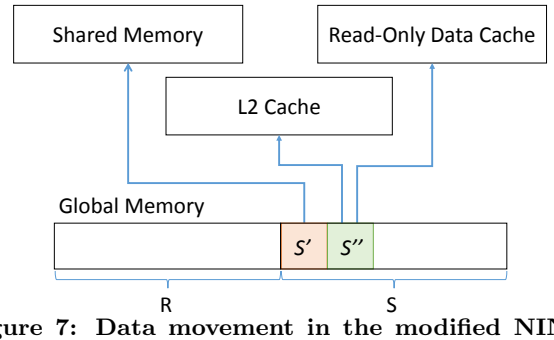

Figure 7: Data movement in the modified NINLJ algorithm. S' and S" are two blocks of table S

5.1 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 7. 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''.

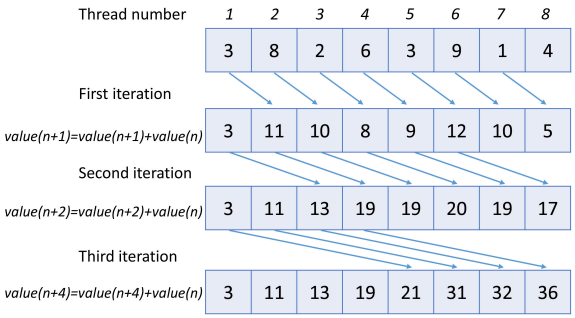

Figure 8: Data access pattern using shuffle instructions

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 8), 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.

5.2 Performance Evaluation

Table 5 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 2), **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. This is surprising to us as our development only involves adding a few lines of code yet the number of registers per thread increased a lot – this clearly shows there are still room for improvement in the CUDA compiler. One thing to point out is: in all above tests, the profiler shows that the utilization of read-only/L2 cache increases as exactly what we expect – this validates our implementation.

Table 5: 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 6 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. The 1.52X speedup is without doubt a significant boost of performance – it increases the Titan/i7 speedup shown in Figure 4 to 13.68X. 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 6: 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

6. 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 run experiments to evaluate energy/power efficiency, floating point computing performance, database size, and kernel configuration. Most of the issues are mentioned in [20] but without much quantitative results.

6.1 Energy / Power Consumption

Energy consumption has become a first-class performance goal in modern computing system design. Energy has become the second largest cost in maintaining today's IT infrastructure [15]. In a data center environment, it also has profound effects on the design and operational cost of cooling systems. With the energy efficiency of CPUs steadily increasing in the last decade, the GPU industry is also making every effort to make GPUs green. The thermal design

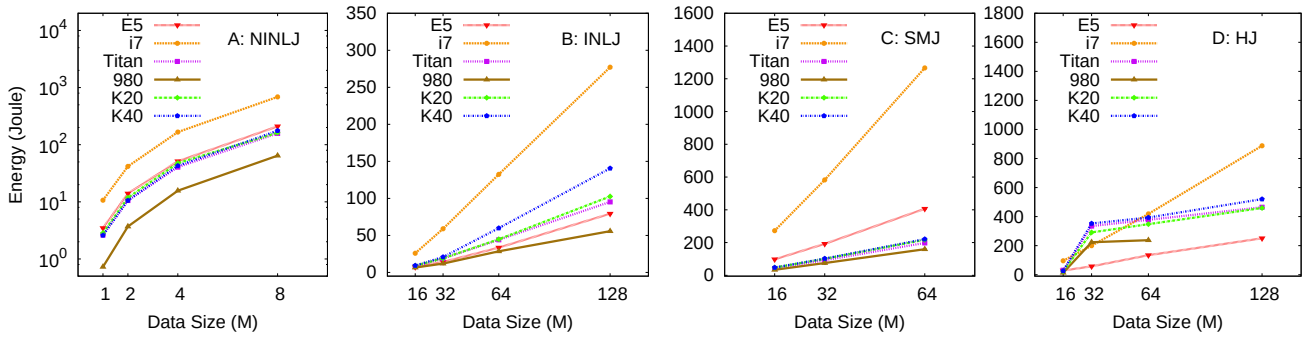

Figure 9: Energy consumption of different CPUs/GPUs in processing four join algorithms

power (TDP) information listed in Table 1 is a good indication of the typical level of power the hardware runs on. By combining TDP and performance-related hardware specifications (i.e., memory bandwidth, maximal Gflops) in Table 1, we can get a rough idea on the energy efficiency of different hardware. Clearly, with less than three times of power consumption yet many times of bandwidth and computing capacity as those of the CPUs, the GPUs stand out as the winners on paper.

Table 7: 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	–

To study how the above insights derived from hardware specifications are reflected in join operations, 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 2, the average active power consumption is shown in Table 7. 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 9). 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 Table 16.

6.2 Floating Point Performance

So far we have only tested the integer computing capabilities of GPUs. Although the most expensive part of the join algorithms is memory access and the computation part only involves simple key value comparison, it is worthwhile to study how GPU performs on processing joins with key data types other than integers. In [20], this aspect was listed as a weakness of GPGPU as high precision numbers were not supported by any GPUs at that time.

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 Fermi architecture boosts its double precision performance by 4.2X over the predecessor architecture. 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 10 shows the

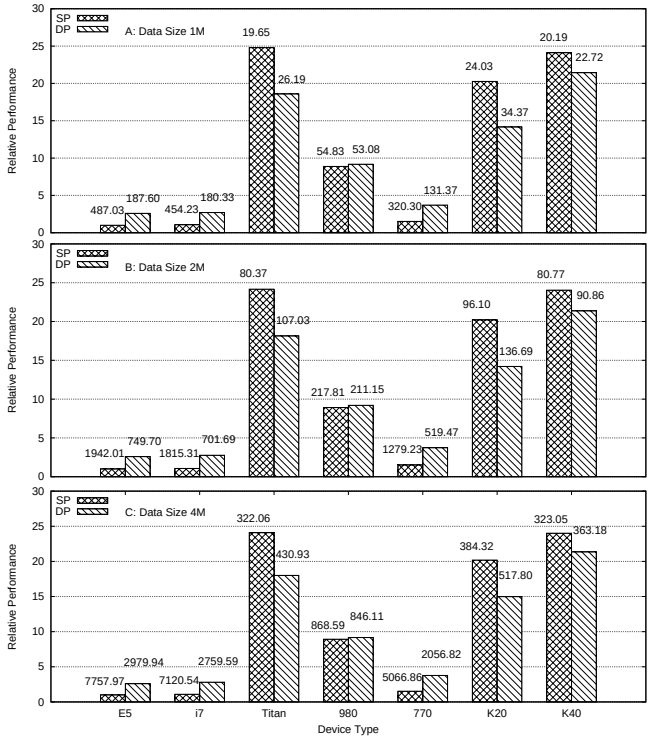

Figure 10: Relative performance of NINLJ with SP/DP keys in different CPUs/GPUs

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 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 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 GTX 980 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.

The above results are also carried over to energy consumption. According to Table 8, the power of computing joins with SP and DP keys in GPUs is 20-30% lower than that with integer keys! For the CPUs, slightly lower power is observed in SP keys while up to 25% more power is recorded in DP keys. This amplifies the advantages of GPUs when energy consumption is calculated (Table 16). The GTX Titan stands out as the clear winner in all cases, followed by the GTX 980 and then the E52640. Without surprise, the i7 is the least energy efficient one among the four devices. For SP, the energy efficiency of GPUs is 3.9-6.8X as high as that of CPUs; For DP, this number is 1.7-3.0X.

Table 8: Active power consumption (watt) of NINLJ with floating-point key values

Quantity	Table Size	Xeon E5	Core i7	GTX Titan	GTX 980
Average(SP)	1M	24.57	90.79	115.05	110.38
	2M	25.57	90.41	151.34	113.04
	4M	26.44	92.73	163.30	115.54
Average(DP)	1M	32.36	98.07	157.58	110.17
	2M	34.72	102.96	147.38	116.88
	4M	36.12	102.43	146.22	115.53
Peak(SP)	1M	25.28	94.59	228.70	121.85
	2M	26.15	95.50	224.10	118.59
	4M	37.95	98.40	209.94	120.20
Peak(DP)	1M	32.82	103.86	197.94	119.04
	2M	37.70	108.28	209.63	172.87
	4M	46.43	111.93	229.30	119.12

6.3 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 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.

6.4 Parameters for Launching a Kernel

Another issue discussed in [20] is the configuration of kernel launching parameters and their effects on kernel performance. Such a problem is of high practical value – a model that accurately predicts kernel performance under different parameters can help us in: (1) identifying the set of parameters leading to optimized performance; and (2) controlling the performance (towards a balanced load) of individual kernels in a multi-kernel environment. In [20], the Map and Split kernels were launched under various numbers of blocks (NoB) and numbers of threads per block (NoTB) and their running time recorded (Section 5.3 in [20]). Their experimental results identified two suitable numbers for each of the two variables mentioned above. The results regarding NoB (Figure 8 in [20]) are straightforward: larger number of blocks means higher level of parallelism till it reaches a point where there are too many blocks than what the hardware

can schedule concurrently. For NoTB, however, no conclusions were drawn other than some qualitative discussions on suitable numbers shown in their experiments.

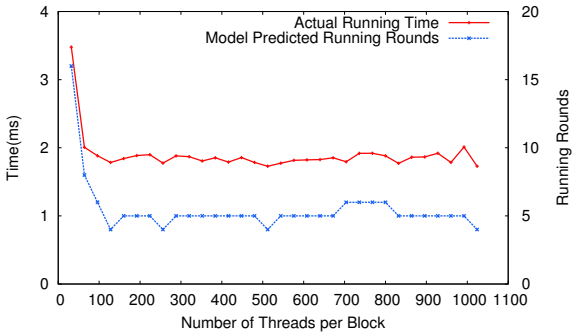

Figure 11: Model-predicted and actual performance of the NINLJ kernel under different NoTB values

With the improvement of GPGPU techniques and more information about the CUDA runtime environment revealed by NVidia, using a quantitative model to predict kernel performance becomes feasible. Actually, in our recent research we developed a model using an indicator of the level of parallelism achieved called *Running Rounds* to predict the performance of compute-intensive single kernels under different NoTB values [27]. Here we just present the validation of our model in the context of join algorithms instead of elaborating on the modeling details (interested readers can refer to [27] for such details). Specifically, we run the NINLJ kernel under different NoTB values, measure the actual running time and compare it with the running rounds provided by our model. From Figure 11, we can see the measured running time (i.e., blue line) matches the running rounds (i.e., red line) given by our model very well. The Pearson correlation coefficient between the two lines is 0.9606, indicating a near-perfect matching. Our on-going work in this direction involves modeling the performance of pure memory-bound kernels (e.g., *map* and *split*) as well as controlling kernel performance by changing total number of threads (in addition to NoB and NoTB). Due to space limit and double submission issues, we skip such details in this paper.

7. 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 [20].

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 [29] 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 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 [5].

As to program development, it is still true that GPU code has to be written from scratch [4] 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 5) 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.

Developers now have abundant choices of GPU devices ranging from entry-level integrated gaming cards to professional compute-dedicated cards. Our work show that a mainstream or high-end gaming card is adequate for general application development. It often costs a few hundred US dollars but delivers desirable performance. If the code has higher demand on double-precision performance and reliability (e.g., ECC memory), more expensive cards (i.e., K20, K40) can be considered. We have not experienced any reliability issues in the join code we tested.

Gaming GPUs are often equipped with relatively smaller global memory (typically 2GB VS 8GB), and the memory on GPU is not upgradable because they are soldered on the board. Data larger than the GPU's memory size requires frequent data transfer between host and device. This undermines the capability of GPU in that the overhead of sending data back and forth between host and device increases as the size of data grows. Fortunately, the GPU vendors seem to be changing the situation. In our test bed, the GTX Titan with 6GB memory can hold two tables each with 250 million tuples in non-index nested loop join, which result in 4GB data plus index storage and other temporary variables. The GPUs can store much more data than they did a few years ago.

[4]OpenCL follows a heterogeneous programming model that supports multi-core CPUs and GPUs, but it still requires hardware-specific program development for code optimization.

Response to concerns shown in [20]: 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 does pose a threat to GPU join performance. As an SIMD architecture, a GPU can suffer from 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 large amount of resources in modern GPUs, memory stall can be more 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. The problem is that dynamic memory allocation is not allowed in CUDA at this moment.

We also appreciate the modularized design of the join programs, in which a series of primitives are implemented as GPU kernels. In our experience, the amount of work in reimplementing the program can be greatly reduced by such. For example, the work reported in Section 5 only involves modifying two kernels, in which only a few lines of code are added/modified.

The program development environment on GPUs has been improved dramatically in recent years. For example, a comprehensive set of tools are provided as part of the CUDA SDK. With such tools, programmers can develop and debug their applications as (if not more) conveniently as one can do with CPUs. In particular, we find the CUDA Visual Profiler very handy in visualizing the runtime statistics of CUDA kernels. We routinely use it to find possible bottlenecks in the application, and optimize such based on suggestions it provides. Such tools fundamentally change the way we develop program from pure empirical experimentation to guided analysis based on quantitative measurements. Our work in modeling performance under different kernel parameters (Section 6.4) is another example that shows the value of the improved software support.

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 6.1). It might be infeasible to expect that future GPUs carry the same power tag as CPUs, considering the huge difference of computing capability between the two. Plus, 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 the performance of joins with SP and DP keys is many times higher than that in the CPUs (Section 6.2).

8. 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

This project is supported by a Faculty Early Career Development (CAREER) Award (No. IIS-1253980) from the US National Science Foundation (NSF) and a gift from Nvidia Corporation via the CUDA Research Center program. The authors want to express their sincere gratitude to Prof. Bingsheng He, who generously provided the code and detailed instructions that made this study possible.

9. 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] International Technology Roadmap for Semiconductors, 2002. http://public.itrs.net.

[12] 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.

[13] 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.

[14] S. Blanas, Y. Li, and J. M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 37–48, 2011.

[15] A. P. Conversion. Determining Total Cost of Ownership for Data Centers and Network Room Infrastructure, 2005. ftp://www.apcmedia.com/salestools/CMRP-5T9PQG_R2_EN.pdf.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] K. Kato and T. Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Procs. of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 769–773, 2010.

[25] N. Kumar, S. Satoor, and I. Buck. Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. In *Procs. IEEE Intl. Conf. on High Performance Computing and Communications (HPCC)*, pages 103–109, June 2009.

[26] P. Kumar, B. Ozisikyilmaz, W.-K. Liao, G. Memik, and A. Choudhary. High performance data mining using r on heterogeneous platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1720–1729, May 2011.

[27] H. Li, D. Yu, A. Kumar, and Y.-C. Tu. Performance Modeling in CUDA Streams - A Means for High-Throughput Data Processing. In *Procs. 2nd IEEE International Conference on Big Data*, 2014.

[28] Y. Liu, X. Liu, and E. Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *Proc. 12th Pacific Conf. Computer Graphics and Applications, PG'04*, pages 247–256, Oct 2004.

[29] 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.

[30] J. Meredith, S. Alam, and J. Vetter. Analysis of a computational biology simulation technique on emerging processing architectures. In *Procs. IEEE Intl. Parallel and Distributed Processing Symposium, IPDPS 2007*, pages 1–8, March 2007.

[31] 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.

[32] 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.

[33] 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.

[34] C. Yang, Q. Wu, J. Chen, and Z. Ge. GPU Acceleration of High-Speed Collision Molecular Dynamics Simulation. In *Procs. 9th IEEE International Conference on Computer and Information Technology, CIT '09*, volume 2, pages 254–259, Oct 2009.

[35] 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.

[36] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Procs. 1991 ACM/IEEE Conference on Supercomputing*, SC '91, pages 712–721, 1991.

[37] M. Zechner and M. Granitzer. Accelerating K-Means

on the Graphics Processor via CUDA. In *Procs. Intl. Conf. Intensive Applications and Services, INTENSIVE '09*, pages 7–15, April 2009.

[38] 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.

APPENDIX

A. PERFORMANCE OF OTHER GPU DEVICES

In this section, we sketch the performance of three other Nvidia GPU cards we tested with the aforementioned join programs. They are: the GTX 770, Tesla K20, and Tesla K40. Note that the GTX 770 is a low-end gaming card while the K20 and K40 belong to the Tesla series that Nvidia promotes as their hard-core compute workhorse. With a very high price tag, the K20 and K40 are frequently chosen to build many of the world's most powerful supercomputers. Specifications of such devices are listed in Table 9. We compare the performance of the three GPUs with the GTX 980 and GTX Titan in Table 10, with a larger number representing better relative performance. The results of NINLJ is very clear, all three GPUs have only a small fraction of the performance of that of the GTX 980 and Titan. In INLJ, K40 outperforms the GTX 980 and Titan by a small margin under smaller table sizes. For SMJ, K40 wins over GTX 980 but is outperformed by Titan. For the join-related primitives (Table 11), the K40 is again outperformed by GTX 980 in all cases (with the map primitive being the only exception). As compared to Titan, K40 performs better in three primitives: map, prefix scan, and qsort.

The floating-point performance of the three GPUs is shown as part of Figure 10. We can easily see that the SP performance of K40 is essentially the same as that of the GTX Titan, the K20 is about 20% slower than the K40, and the GTX 770 is the worst, with a performance only comparable to that of the two CPUs. For joins with DP keys, K40 overperforms the GTX Titan by a 10-15% margin. Again, it is tailed by the K20, and GTX 770 only achieves a performance that is slightly better than the CPUs.

The overall energy efficiency of the three GPUs as compared to GTX 980 and Titan is presented in Table 12, in which a larger number represents lower energy efficiency. If we consider integer-type join keys, all three GPUs tested here are much less energy efficient than the GTX 980 but stay on the same level as that of the Titan. In fact, for most cases of HJ, the GTX 770 and K20 consume less energy than the Titan. By considering floating-point computation, the K20 and K40 is much more energy efficient than the GTX 980 (in NINLJ) but are still less energy efficient than the Titan. The energy efficiency of GTX 770 under SP and DP drags far behind all other GPU devices.

In summary, the Tesla series cards, including both K20 and K40, do not deliver better performance in processing joins than the less expensive gaming cards. In some cases, their performance even fall far behind to that of the GTX 980 and Titan. The top-of-the-line K40 outperforms the latter two by a small margin in several cases. The same trend is observed in energy efficiency. Their floating-point performance seems to be in par with the gaming cards. The GTX 770, being a low-end card, does not surprise us with its mediocre performance and energy efficiency.

Table 9: Specifications of more hardware

Device	GPU		
	Tesla K40	Tesla K20	GTX 770
Date Released	Q4 2013	Q4 2012	Q2 2013
Core Speed	0.75GHz	0.71GHz	1.05GHz
Core Count	15×192	13×192	8×192
Cache Size	L1: $64KB \times 15$ L2:1536KB	L1: $64KB \times 13$ L2:1280KB	L1: $64KB \times 8$ L2:512KB
RAM	GDDR5 12GB 384 bit	GDDR5 5GB 320 bit	GDDR5 2GB 256 bit
Memory Bandwidth	288GB/s	208GB/s	224GB/s
Max GFLOPS	4291	3524	3213
Max TDP	245W	225W	230W
Launch Price	7699 USD	3199 USD	399 USD

Table 10: Relative performance of four join algorithms on different GPUs

Algorithm	Data Size	GPU to GPU Speedup					
		K40/980	K20/980	770/980	K40/Titan	K20/Titan	770/Tian
NINLJ	1M	0.38	0.29	0.33	0.92	0.70	0.81
	2M	0.38	0.29	0.33	0.97	0.73	0.85
	4M	0.38	0.29	0.34	0.99	0.75	0.87
	8M	0.38	0.29	0.34	1.01	0.76	0.89
INLJ	16M	1.09	0.74	0.68	1.14	0.78	0.71
	32M	0.88	0.66	0.59	1.00	0.76	0.67
	64M	0.76	0.65	0.67	0.88	0.76	0.78
	128M	0.76	0.66	–	0.87	0.76	–
SMJ	16M	1.06	0.75	0.84	0.99	0.70	0.79
	32M	1.06	0.76	0.85	0.98	0.70	0.79
	64M	1.06	0.75	–	0.98	0.70	–
HJ	16M	0.76	0.48	0.53	1.27	0.81	0.88
	32M	0.90	0.70	0.19	0.99	0.77	0.21
	64M	0.88	0.67	–	1.02	0.78	–
	128M	–	–	–	1.04	0.77	–

Table 11: Relative performance of join-related data primitives between GPUs

Algorithm	GPU to GPU Speedup					
	K40/980	K20/980	770/980	K40/Titan	K20/Titan	770/Titan
map	1.00	0.67	0.54	1.50	1.00	0.62
scatter	0.80	0.66	1.04	0.97	0.80	1.25
gather	0.90	0.75	0.64	0.70	0.58	0.50
prefix scan	0.80	0.57	0.50	1.20	0.86	0.75
split	0.79	0.66	0.56	0.70	0.58	0.49
qsort	0.87	0.66	0.59	1.02	0.78	0.70

Table 12: Relative active energy consumption of four algorithms between GPUs

Algorithm	Table size	770/980	K20/980	K40/980	770/Titan	K20/Titan	K40/Titan
NINLJ	1M	3.37	3.91	3.56	0.93	1.08	0.99
	2M	2.73	3.19	2.85	0.96	1.12	1.00
	4M	2.67	3.04	2.73	1.03	1.17	1.05
	8M	2.72	2.52	2.73	1.11	1.03	1.12
INLJ	16M	1.54	1.38	1.41	1.27	1.14	1.17
	32M	1.81	1.59	1.74	1.15	1.01	1.11
	64M	1.51	1.57	2.10	0.98	1.02	1.36
	128M	–	1.84	2.53	–	1.08	1.48
SMJ	16M	1.43	1.41	1.42	1.16	1.14	1.15
	32M	1.42	1.35	1.37	1.16	1.09	1.11
	64M	–	1.36	1.39	–	1.10	1.12
HJ	16M	1.65	1.75	1.95	0.72	0.77	0.85
	32M	0.95	1.30	1.58	0.63	0.87	1.05
	64M	–	1.47	1.66	–	0.92	1.04
	128M	–	–	–	–	0.99	1.12
NINLJ(SP)	1M	6.12	0.51	0.58	16.40	1.38	1.56
	2M	6.11	0.51	0.59	12.36	1.03	1.19
	4M	6.03	0.51	0.60	11.52	0.97	1.14
NINLJ(DP)	1M	2.45	0.81	0.67	3.47	1.15	0.94
	2M	2.38	0.78	0.64	3.73	1.23	1.01
	4M	2.45	0.77	0.68	3.80	1.19	1.05

B. MORE EXPERIMENTAL DATA

The following tables show more experimental results that we have discussed in the main body of the paper.

Table 13: NINLJ performance (sec) using SP keys

Data Size	E5 2640	i7 3930k	GTX Titan	GTX 980	980 vs E5	Titan vs i7
1M	487	454.2	19.7	54.8	8.9	23.1
2M	1942	1815.3	80.4	217.8	8.9	22.6
4M	7758	7120.5	322.0	868.6	8.9	22.1

Table 14: NINLJ performance (sec) using DP keys

Data Size	E5 2640	i7 3930k	GTX Titan	GTX 980	980 vs E5	Titan vs i7
1M	187.6	180.3	26.2	53.1	3.5	6.9
2M	749.7	701.7	107.0	211.2	3.6	6.6
4M	2979.9	2759.6	430.9	846.1	3.5	6.4

Table 15: Peak active power consumption (watt)

Algorithm	Table Size	Xeon E5	Core i7	GTX Titan	GTX 980
NINLJ	1M	28.60	99.99	199.01	169.82
	2M	29.14	100.43	230.58	180.94
	4M	27.30	143.69	233.63	189.91
	8M	28.06	103.41	234.32	217.97
INLJ	16M	31.20	90.58	84.58	91.52
	32M	38.84	111.80	105.98	119.34
	64M	39.61	109.80	130.03	103.34
	128M	39.26	108.52	177.19	110.18
SMJ	16M	11.37	92.13	135.76	103.69
	32M	11.46	64.25	186.07	135.92
	64M	19.80	115.71	200.49	127.68
HJ	16M	10.35	70.22	108.74	78.37
	32M	10.81	63.41	125.23	126.59
	64M	11.93	64.08	136.51	101.23
	128M	11.89	63.94	165.72	–

Table 16: Active energy consumption of four algorithms running on different hardware

Algorithm	Table Size	Energy Consumed (Joule)				Relative Energy	
		E5-2640	i7-3930K	GTX Titan	GTX980	i7 / 980	E5 / Titan
NINLJ	1M	3.47×10^3	1.07×10^4	2.63×10^3	727.85	14.67	1.32
	2M	1.40×10^4	4.17×10^4	1.06×10^4	3.70×10^3	11.27	1.32
	4M	5.13×10^4	1.66×10^5	4.05×10^4	1.57×10^4	10.61	1.26
	8M	2.09×10^5	6.87×10^5	1.58×10^5	6.44×10^4	10.68	1.33
INLJ	16M	7.25	25.93	8.00	6.62	3.92	0.91
	32M	14.14	58.98	18.81	11.95	4.94	0.75
	64M	33.53	132.42	44.07	28.53	4.64	0.76
	128M	79.31	277.15	95.29	55.70	4.98	0.83
SMJ	16M	97.67	272.68	41.60	33.65	8.10	2.35
	32M	192.16	582.71	92.17	74.92	7.78	2.08
	64M	406.28	1266.11	197.82	159.28	7.95	2.05
HJ	16M	28.76	95.80	30.54	13.37	7.16	0.94
	32M	57.52	199.67	335.01	223.41	0.89	0.17
	64M	134.12	418.48	377.62	237.40	1.76	0.36
	128M	250.71	887.58	464.57	–	–	0.54
NINLJ(SP)	1M	1.20×10^4	4.12×10^4	2.26×10^3	6.05×10^3	6.81	5.29
	2M	4.97×10^4	1.64×10^5	1.22×10^4	2.46×10^4	6.67	4.08
	4M	2.05×10^5	6.60×10^5	5.26×10^4	1.00×10^5	6.58	3.90
NINLJ(DP)	1M	6.07×10^3	1.77×10^4	4.13×10^3	5.85×10^3	3.02	1.47
	2M	2.60×10^4	7.22×10^4	1.58×10^4	2.47×10^4	2.93	1.65
	4M	1.08×10^5	2.83×10^5	6.30×10^4	9.78×10^4	2.89	1.71