GPU Join Processing Revisited

Tim Kaldewey[§] Guy Lohman[§] Rene Mueller[§] Peter Volk[†*]
[§]IBM Almaden Research, San Jose, CA
[†]Technische Universität Dresden, Dep. of Computer Science
{tkaldew, lohmang, muellerr}@us.ibm.com peter.volk@tu-dresden.de

ABSTRACT

Until recently, the use of graphics processing units (GPUs) for query processing was limited by the amount of memory on the graphics card, a few gigabytes at best. Moreover, input tables had to be copied to GPU memory before they could be processed, and after computation was completed, query results had to be copied back to CPU memory. The newest generation of Nvidia GPUs and development tools introduces a common memory address space, which now allows the GPU to access CPU memory directly, lifting size limitations and obviating data copy operations. We confirm that this new technology can sustain 98 % of its nominal rate of 6.3 GB/sec in practice, and exploit it to process database hash joins at the same rate, i.e., the join is processed "on the fly" as the GPU reads the input tables from CPU memory at PCI-E speeds. Compared to the fastest published results for in-memory joins on the CPU, this represents more than half an order of magnitude speed-up. All of our results include the cost of result materialization (often omitted in earlier work), and we investigate the implications of changing join predicate selectivity and table size.

1. INTRODUCTION

Memory bandwidth exceeding 150 GB/s and hundreds of cores make GPUs an interesting platform for accelerating complex query processing tasks such as joins. Nvidia's *Compute Unified Device Architecture (CUDA)*, an extension of the C programming language, simplifies programming GPUs for applications other than graphics [6]. However, the use of GPUs for data-intensive operations has been limited by the amount of memory on the GPU card (≤6GB today) and the time-consuming process of copying data back and forth between CPU and GPU memory across a *PCI Express* (PCI-E) link with limited bandwidth (≤6.3 GB/s today).

Though earlier investigations exploring the use of GPUs for join processing claimed orders of magnitude speedup over CPUs [7–9], they were based on assumptions that sidestepped reality. They assumed that both the input tables *and* the join results all fit simultaneously in the GPU's limited memory. Often, they only measured the time to perform the join, omitting the non-trivial transfer times from and to CPU memory or considering them negligible, which we found not to be the case for efficient join implementations. While it was feasible to work around the memory limitation by partitioning the input tables, overlapping data copies and processing to effectively use the available PCI-E bandwidth has proven challenging. Recent work [12] on offloading hash probes to the GPU identified data copying as the dominant cost (≥ 50 %) that limited effective throughput to 50 % of the available PCI-E bandwidth.

The latest generation of Nvidia GPUs and development tools adds a common address space for the CPU and GPU called *Unified Virtual Addressing* (UVA), which allows the GPU to access the CPU-side memory directly. This not only lifts the size limitations on data sets that can be processed, but also relieves the programmer from the burden of managing two address spaces and copying data back and forth. More importantly, UVA can sustain 98 % of the nominal rate of PCI-E; we measured 6.2 GB/s in practice. UVA enables the GPU to process arbitrarily large tables in CPU memory at PCI-E speeds, without managing data transfers or multiple copies.

This paper shows how UVA can be leveraged to accelerate database join processing, achieving throughput rates of up to 6.1 GB/s, making optimal use of the available PCI-E bandwidth (Sec. 3). This amounts to a speedup of more than half an order of magnitude compared to previously published results for in-memory join operations on the CPU [10]. We provide an end-to-end performance analysis of relational joins on GPUs, i.e., including the cost of reading the input data sets from CPU memory and materializing results. Our GPU results concur with prior work on in-memory joins on the CPU [3] that a conventional (non-partitioned) hash join works best, due to its simplicity. We also analyze the sensitivity of our implementation to table size and join predicate selectivity. Our results show that the table size has marginal impact upon throughput, whereas selectivity of the join predicate significantly affects performance (Sec. 4). We propose a result cache to accelerate joins that produce a large number of results. Since the performance of database operations is known to be dominated by memory performance [1, 4], we provide a detailed evaluation of GPU memory performance exploiting UVA (Appendix A, B).

2. GPU BACKGROUND

This section provides a brief tutorial on GPU architecture

*Work done while the author was at IBM

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

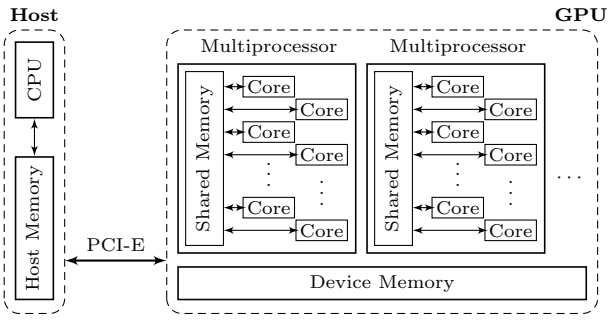


Figure 1: Architecture of an Nvidia GPU.

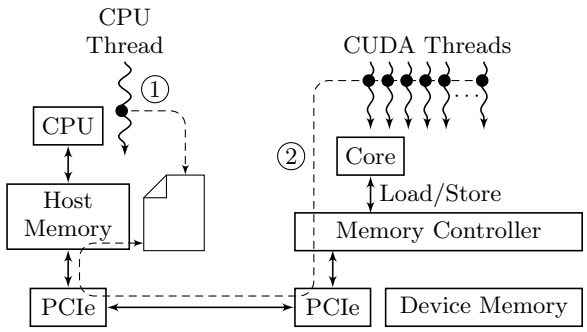


Figure 2: UVA Read access from the GPU Kernel into host memory. (1) CPU thread pins page. (2) CUDA threads execute a load or store instruction, which results in a memory read or write transaction on the PCI-E link to the pinned page.

and the new UVA enhancement, as it pertains to database joins. For a more detailed discussion of relevant memory access patterns, please refer to the Appendix.

A GPU application written for CUDA consists of two parts: a "control" thread running on the *host* (CPU) and a parallel part, called a *kernel*, running on the GPU. The kernel is organized as a number of *thread blocks*, with each block running all its threads on the *CUDA cores* of one *streaming multiprocessor* (SM), see Figure 1. CUDA cores within the same SM execute the same instruction in lockstep, in *Single Instruction-Multiple Data* (SIMD) mode. Threads within the same thread block can access a low-latency on-chip *shared memory* on an SM. For example, the GTX 580 GPU we use in our experiments has 16 SMs, and each SM has 32 CUDA cores and 48 kB of shared memory.

Host memory access. Data is transferred between the host memory and the GPU over a PCI-E link, which puts an upper bound on the throughput of data-intensive operations. Before UVA was introduced with Nvidia's Fermi architecture and CUDA 4.0, this had to be done through explicit host-initiated `cudaMemcpy()` calls, hereafeter referred to as `memcpy`. Now, UVA allows a compute kernel to directly access host memory, which is fetched over the PCI-E link as needed. The concept of UVA is illustrated in Figure 2. In step (1), the CPU thread of a CUDA application pins a page in host to obtain a real address pointer to this page from the operating system. This real address pointer is then passed to the GPU during the kernel invocation. In step (2), the CUDA threads can now issue load and store instructions using this real address pointer. The memory controller on the GPU distinguishes this access from requests to the device memory and performs the corresponding memory read or write transaction over the PCI-E link. The *PCI Express Root Complex* on the host side then issues an access to the host memory. The 16× PCI-E 2.x connection of our GPU has a nominal bandwidth of 6.3 GB/s. We measured an effective bandwidth of 6.2 GB/s for 64-bit read and write accesses to host memory through UVA. Our measurements show that at least 1,024 threads and 16 blocks are required (Appendix B), which places a lower bound on the parallelism required for an efficient hash join implementation.

Device memory access. Hash joins are known to produce irregular memory access patterns during parallel hash table creation and probing. In particular, compare-and-swap for inserting tuples into the hash table and quasi-random, data dependent reads for hash table probes are problematic. On our GTX 580 GPU we measured 7.7 GB/s for 64-bit random read accesses to 512 MB of device memory. For 64-bit compare-and-swap accesses to random locations we measured 4.6 GB/s. To achieve these bandwidths the compute kernel needs to be launched with at least 16 thread blocks and at least 8 threads/block (Appendix A).

3. IMPLEMENTATION APPROACHES

To determine the optimal join algorithm for GPUs, we ported the best known in-memory join algorithms from recent studies of CPU joins: a traditional hash join [3] and a partitioned variant of hash join. The latter algorithm partitions the hashed values of the input keys in such a way that each partition fits into the processor cache, to avoid random accesses to memory when joining each partition [10]. We then evaluated both hash algorithms, first explicitly copying data from and to the CPU, and secondly exploiting UVA. Our GPU implementation of a conventional (parallelized) hash join using UVA outperformed all other approaches and achieved half an order of magnitude speed-up over the partitioned CPU hash join.

Implementation Details. Our parallel join implementations on the GPU do not differ fundamentally from their CPU counterparts. The core algorithms are identical, but the parallelization, data placement, and intermediate data structures are GPU-specific. For example, in the partitioned hash join, the GPU's 48 kB of shared memory takes over the role of the CPU 's caches. The remaining parameters are the same as in prior work [3, 10] to allow performance comparisons. For example, we use the least significant bits (LSB) of the input key as the hash function, and an open addressing hash table two times the size of the input table.

Similar to a conventional CPU hash join implementation, we create a hash table from the *build table* and then probe it with the entries from the *probe table*. As in earlier work [2,9], we build the hash table in the device memory of the GPU. As discussed in Section 2, achieving high throughput requires a high degree of thread parallelism. For example, accessing the input tables located in host memory requires at least 1 k threads to be efficient. Obviously, this requires coordinating write access to shared data structures. We use atomic operations to efficiently coordinate multiple threads concurrently inserting data into the hash table, e.g., compare-and-swap, and into the result set, e.g., an index that is incremented atomically.

Using pre-UVA techniques that perform `memcpy` opera-

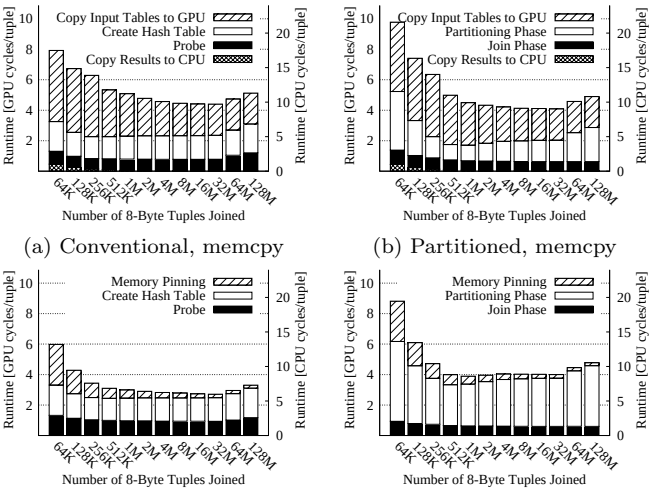
tions, a conventional hash join algorithm copies the build table from host to device memory, creates a hash table in device memory, deletes the build table from device memory, copies the probe table to device memory, probes the hash table, stores the results in device memory, and finally copies them back to host memory. This approach is limited by the combined size of its working data, i.e., the build table (or horizontal partition thereof) and its hash table during build, and during probe the hash table, probe table (or horizontal partition thereof), and results all must simultaneously fit into device memory. It also requires passing control of the program execution back and forth between the GPU and CPU, as `memcpy` operations are controlled by the CPU.

Using UVA, the GPU can read the build table directly from host memory while it creates the hash table in device memory, so it never has to store the build table in the device memory. The probe phase reads the rows of the probe table directly from host memory, probes the hash table stored in device memory, and again stores the results directly in host memory. The only limiting factor is the size of the hash table, which has to fit into device memory ($\leq$6 GB). Just as a CPU join may have to spill portions of its hash table to disk if it exceeds memory, the GPU may have to spill its hash table to host memory using UVA. Implementing support for larger hash tables and evaluating the impact of spilling on performance is future work.

The partitioned hash join described by Kim et al. [10] can also be ported to the GPU with relatively few modifications. The histogram(s) that are used to determine the target location of data in the partitioned data set can be created in shared memory to avoid the performance penalties of frequent device memory accesses. The size of shared memory (48 kB) limits the number of partitions that can be processed in one pass to 12 k (48 kB / 4 bytes per histogram entry) and the partition size to 16 kB (assuming a conservative 50% load factor for the hash table). Therefore, tables larger than 192 MB require multi-phase partitioning.

Data Sets. To compare our results with prior work [3, 10], we join two equally sized tables, with tuples comprised of a 32-bit key and a 32-bit row identifier. Choosing two tables of equal size represents the worst case scenario for a hash join, as the hash table is usually created from the smaller of the two tables, to optimize memory consumption and performance. We use a uniformly distributed, randomly generated data set, again the worst case scenario, as there will be no locality of reference.

We control the result size by varying the *match rate* of the join predicate, which is defined as the percentage of keys in the probe table that have matching key(s) in the build table. [1] Low match rates correspond to joining domains that barely intersect, whereas referential integrity would guarantee a 100 % match rate. We generate the data set such that for each tuple in the probe table, we either randomly select a tuple from the build table or generate a random value that is outside of the key space, according to the match rate. If both tables were simply filled with randomly-generated 32-bit integers, as done by some papers, the match rate could not be controlled, and would be unrealistically low, based upon the probability of generating the same random 32-bit integer in both tables. Other than one experiment that explicitly examines the impact of varying the match rate on performance, our evaluation uses a match rate fixed at 3 %, to be compatible with prior work [10], though we believe it to be unrealistic.



(a) Conventional, memcpy (b) Partitioned, memcpy

(c) Conventional, UVA (d) Partitioned, UVA

Figure 3: Comparing the runtime of conventional and partitioned hash join implementations, using data copies vs. a uniform address space.

Conventional vs. Partitioned Hash Join. Using the traditional GPU programming approach that requires data copies, conventional and partitioned hash joins achieve nearly identical performance. Figure 3(a) and (b) show the execution time per tuple in GPU cycles (left y-axis) and CPU cycles (right y-axis)[2] for increasing number of tuples joined, i.e., the combined size of the build and probe tables. Though performance is comparable (4–5 GPU compute cycles per tuple) for larger data sets, for smaller data sets (<512 K tuples), the conventional hash join achieves slightly better performance (6–8 GPU cycles/tuple) than the partitioned one (7–10 GPU cycles/tuple). Obviously, the cost of data copies, labeled "Copy Input Tables to GPU" for transferring the input tables and "Copy Results to CPU" for transferring the results, are identical for both algorithms. For data sets larger than about 8 M tuples, the cost of initiating input data transfers is dominated by the PCI-E bandwidth, which we measured at 2 GPU cycles/tuple or 6.1 GB/s. The same applies to copying join results back to host memory, except that it occurs earlier, for data sets larger than 512 k tuples, because we assumed highly selective (3 %) joins. Aside from the cost of copying the input tables, the execution time of a conventional hash join depends on both hash table creation (labeled "Create Hash Table") and probing (labeled "Probe"). The rate of hash table creation is governed by the efficiency of compare-and-swap operations, while the speed of probes is limited by random reads from the device memory (Appendix A). The time to read the input tables from device memory is negligible, as these reads are coalesced and achieve rates above 150 GB/s (Appendix A). The perfor-

[1] Though other papers have called this "selectivity", it is *very* different from selectivity as used in the query optimization literature. We use it only so we can compare our results to prior work.

[2] We include the latter metric to enable a direct comparison with prior work on CPU joins, since there is a more than 2× difference between GPU (1.5 GHz) and CPU (3.4 GHz) clock frequency.

mance of probes is directly correlated with that of random read accesses to device memory, which decreases for data set sizes of 256 MB or more (see Appendix A).

The partitioned hash join described by Kim et al. [10] requires three passes over the input tables, one to create a histogram to compute the partition boundaries, one to move the input data to the target partitions, and one to join the subtables. The first two are part of the "Partitioning Phase" and the third is required for the "Join Phase". As with conventional hash join, the input data can be read from device memory at up to 150 GB/s. However, moving data to the partitions requires random writes, which is the dominating cost of the partitioning phase. On the other hand, all random accesses required by hash table creation and probing during the join phase are to fast shared memory, as partitioning is done so that a partition fits into shared memory. Therefore, the overall cost of the join phase is relatively small (< 1GPU cycle/tuple). The increases in execution time for the partitioned hash join on data sets of 64 M tuples (256 MB per table) and larger are the result of 2-phase partitioning, which requires double the scans(4), but more importantly two (random) rewrites of the data set.

Data Copy vs. UVA. Using UVA's uniform address space obviates data copies, since the input tables remain in host memory and results are written directly to host memory. UVA requires the input tables to be pinned in physical memory. While the (per tuple) cost of pinning the input tables in host memory amortizes over larger data sets (Fig. 3(c),(d)), reading the input tables through UVA directly from host memory puts an upper limit on overall throughput (≤ PCI-E bandwidth).

For a conventional hash join, we observe from Figure 3(c) that using UVA to access the input tables has no impact on hash table creation and negligible impact on probing. Compare-and-swap operations are slower than the PCI-E link, and so remain the limiting factor of hash table creation. However, random accesses to device memory perform slightly better than PCI-E bandwidth, so probing is limited by the PCI-E bandwidth.

The partitioned hash join implementation cannot take advantage of UVA (see Fig. 3(d)), as it requires several passes over the input tables, which with UVA remain stored in host memory. Host memory access is more than an order of magnitude slower than device memory access—6 GB/s vs. 150 GB/s. The overall performance is almost exactly the same as using explicit data copies, with the time previously spent on copying the input data now included in the partitioning phase. We conclude that, in the presence of UVA, partitioning input tables does not provide an advantage over a simple hash join implementation, an observation that conforms with prior results comparing CPU implementations of these algorithms [3].

GPU vs. CPU. Our "conventional" GPU hash join using UVA achieves an end-to-end throughput of roughly 3 compute cycles per tuple (Fig. 3(c)), which is about an order of magnitude faster than results reported from the best known parallel CPU implementation [10] (32 CPU cycles/tuple). However, using cycles per tuple as a metric does not permit an "apples-to-apples" comparison, as clock frequencies differ significantly, e.g., our GPU is clocked at 1.5 GHz, versus our CPU at 3.4 GHz and the 3.2 GHz CPU used in [10]. Allowing for the difference in clock frequency between our GPU and CPU by comparing the vertical scale on the right of Figure 3 (measured in CPU clock cycles per tuple), our GPU implementation is still more than half an order of magnitude faster than the best parallel CPU implementation. [3]

4. EXPERIMENTAL EVALUATION

Having determined the best implementation for our GPU join to be a conventional hash join using UVA to directly access the host memory, we now evaluate the efficiency of that implementation, and how sensitive its performance is to table sizes and the selectivity of the join predicate. We also evaluate the benefit of a result cache to reduce contention when materializing results. To identify inefficiencies or bottlenecks independent of a specific machine's architecture or performance characteristics, for the remainder of this paper we measure performance and compare it to the nominal hardware capabilities in terms of throughput, i.e., the number of input bytes processed per second.

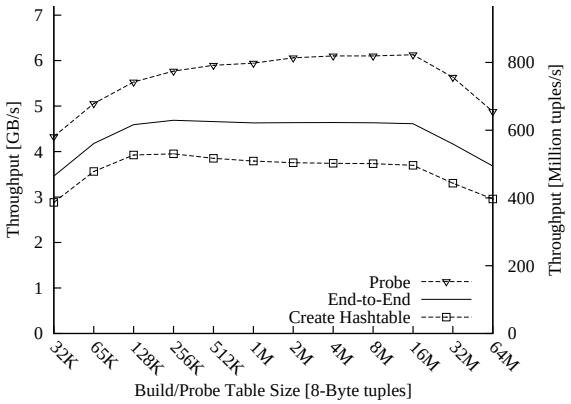


Figure 4: Throughput of GPU hash join with respect to input table size(s), with equally sized input tables accessed via UVA, and match rate fixed at 3 %.

Efficiency. Based on the same experiment as Figure 3(c), Figure 4 depicts the throughput of build phase, probe phase, and overall, end-to-end query execution as a function of the size of the input tables. The end-to-end throughput is the geometric mean of the two stages of our hash join — hash table build and probe. We observe an overall performance of up to 4.6 GB/s, and little sensitivity to the table sizes. Hash table probes can be executed at up to 6.1 GB/s, the maximum rate at which the probe table can be accessed across the PCI-E link (Appendix B), while hash table creation is significantly slower, reaching at most 3.9 GB/s. The limiting factor for parallel hash table creation is the locking required to prevent parallel threads from overwriting existing hash table entries. Our implementation uses compare-and-swap to manage concurrent hash table access (Sec. 3). The 18 % performance difference between the resulting hash table build rate (3.9 GB/sec) and compare-and-swap throughput (4.6 GB/s) can be attributed to the time necessary to initialize the hash table and to resolve hash collisions. In fact, excluding the time it takes to initialize the hash table, we measured hash table creation rates up to 4.1 GB/s, and

[3]Running the partitioned CPU hash join on our quad-core i7 Sandy Bridge CPU, making use of all of its eight hardware threads, we observe a 20% performance improvement over the previous CPU generation [10].

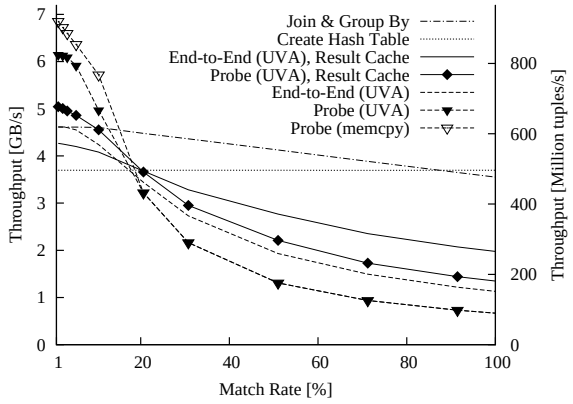


Figure 5: Throughput of GPU hash join as a function of percent of rows finding matches, with equally sized input tables of 16M tuples accessed via UVA

with a perfect hash that produces no collisions[4] we observe rates up to 4.5 GB/s, close to optimal.

Match Rate. The previous experiments used a match rate of 3 %, which made the time to write these few results back to host memory negligible, compared to the time required to read the probe table from memory. Other work on in-memory joins on the CPU suggests that the time required to materialize join results can be neglected, as it only adds a constant overhead of a few compute cycles per result [3, 10, 11]. However, given that our best GPU join implementation requires only 3 cycles per tuple (Fig. 3(c)), a few extra cycles can add substantial overhead when more probes return results.

Figure 5 confirms that join predicates that have a higher percentage of matches significantly reduce GPU join throughput. We ran a conventional hash join, labeled "End-to-End (UVA)", for data sets of 16 M tuples per table accessed via UVA, and varied the match rate from 1 % to 100 %. Though earlier papers assumed a very optimistic match rate of 3 %, the more common situation of a join between a foreign key and a primary key would result in a 100 % match rate. [5] For join predicates having match rates less than 5 %, throughput is high, but beyond this point the throughput drops steeply from 4.6 GB/s to 1 GB/s.

When the join predicate has a low match rate, overall throughput was limited by hash table creation, labeled "Create Hash Table" in Figure 5, but as the match rate increases, throughput becomes dominated by the probe phase, labeled "Probe (UVA) ". Obviously, the join predicate's match rate does not impact hash table creation, so we will limit further investigation to the performance of hash table probes.

Although PCI-E is bidirectional, the GPU we used in our test system can only transfer data in one direction at a time, as it only has one DMA engine. [6] To determine the performance impact of simultaneously probing the hash table and transferring results across the PCI-E link, we repeated the experiment above with input tables stored in device memory and results written to device memory as well. To our surprise, the curve labeled "Probe (memcpy)" in Figure 5 shows that using device memory exclusively provides only marginal improvements over UVA, which indicates that UVA is not a bottleneck here.

Our measurements in Appendix B show that we need to launch at least 1 k threads to make efficient use of the available PCI-E bandwidth. The less selective the join predicate is, the more of these 1 k GPU threads that will find a match and will need to write its result contiguously to host memory at the same time, inevitably creating contention for the write coordinator. Although we coordinate writes as efficiently as possible, using an index on an array that is incremented atomically, as join match rates approach 100 %, 1 k threads access this index simultaneously, creating a bottleneck.

Result Cache. Reducing the number of threads attempting to write results to the same data structure will reduce this contention. To accomplish this, we implemented a result cache that uses on-chip shared memory (cf. Fig. 1) to stage results before they are written to host memory. With this cache, we only need to coordinate the writes of threads within a single block, as only they have access to the same shared memory (on the same SM). Moreover, once the result cache fills up, we use all of those threads to write (flush) the cached results in parallel, which allows for efficient coalesced writes. On the downside, every time we flush the result cache, we need to synchronize all threads within that block before we can write its content back to host memory. Since shared memory and therefore our cache is limited to 48 kB, for joins with high match rates we have to flush the cache more frequently, which again requires an atomic operation on the UVA-accessible memory. Nevertheless, our result cache reduces the pressure on the atomic insert into the result table by more than three orders of magnitude, as we can cache up to 6 k results.

Our result cache doubled the throughputs of probe and overall, labeled "Probe (UVA), Result Cache" and "End-to-End (UVA), Result Cache", respectively, in Figure 5. Although the result cache marginally reduces overall throughput for joins with match rates < 10 %, the result cache improves throughput thereafter, with more gradual decreases from > 4 GB/s for 10 % match rate to 3 GB/s for 50 % match rate. Even for 100 % match rates, our implementation still achieves 2 GB/s end-to-end throughput, which is nearly three times faster than the fastest reported in-memory CPU join [10][7], which assumed a very optimistic 3 % match rate and did not include the cost of result materialization in its performance evaluation.

Operator Pipelining. Prior work on GPU joins [12] suggests that a join is often followed by a group by (and aggregate) operator. Pipelining the join results into the group by operator so that the same set of threads will handle the aggregation as well, avoids materializing the results in host memory and the locking associated with it. The curve labeled "Join & Group By" in Figure 5 depicts a scenario in which each thread computes a local aggregate at its end and is added to a global aggregate located in host memory using an atomic add. For this scenario, overall throughput grad-

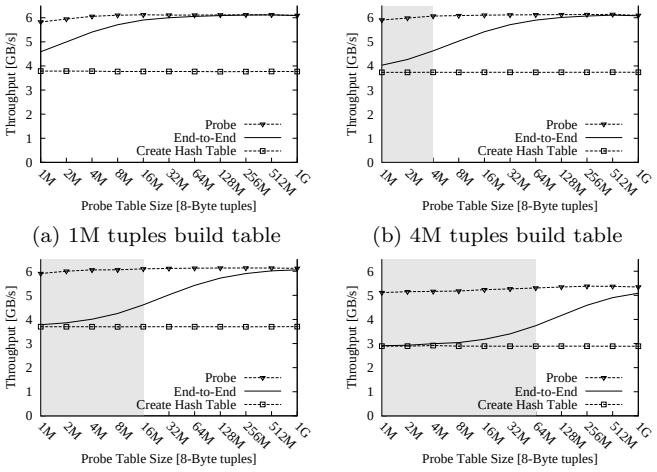
[4]For the LSB hash we are using for our experiments, creating a build table with consecutive even or odd integers as keys satisfies this condition.

[5]We are only considering the effect of the join predicate itself, and assume that any predicates local to individual tables have already been applied.

[6]Professional video cards like Nvidia's Quadro series or specialized GPU compute cards like the Tesla series have two DMA engines enabling bidirectional use of the PCI-E link.

[7]32 cycles/8-byte tuple at 3.2 GHz are the equivalent of 0.75 GB/s.

ually decreases from 4.6 GB/s for match rates of 1–5 % to 3.6GB/s for 100 %.



(a) 1M tuples build table (b) 4M tuples build table

(c) 16M tuples build table (d) 64M tuples build table

Figure 6: Throughput of GPU hash join for increasing input table sizes, tables accessed via UVA and match rate fixed at 3 %.

Table dimensions. While our prior experiments focused on the worst case scenario for (hash) joins, i.e., building and probing tables of equal size, we now take a look at the common case in which the build table is smaller than the probe table [3]. Figure 6 depicts the performance of our GPU join implementation for build tables of size 1, 4, 16, and 64 million tuples, with increasing probe table size from one million to one billion tuples, and a constant match rate of 3%. The rates of hash table creation (labeled "Create Hash Table") and probe (labeled "Probe") are virtually constant at 3.9 GB/s for building hash tables for 1 M, 4 M, 16 M tuples, and 6 GB/s for probes, independent of the probe table size. The overall throughput (labeled "End-to-End") is the geometric mean of the two, weighted by the number of tuples. Hence, with increasing probe table size, performance becomes dominated by probing.

Figures 6(c) and (d) leave the impression that for smaller probe tables, performance is dominated by hash table creation. This is an artifact of using the same range for probe table sizes across all experiments (a–d), and always creating the hash table based on the table identified as the build table, regardless of size. Typically, an optimizer would choose the smaller table to build the hash table, both to save memory and because the hash table creation rate is slower than the probe rate. For example, building a hash table on a 16 M-tuple table and probing it with a table with 1 M tuples reduces overall throughput to the rate of hash table creation, 3.9 GB.s, as shown in Fig. 6(c). Conversely, building a hash table on the 1 M-tuple table and probing it with the table with 16 M tuples yields far better performance of 6 GB/s, as shown in Fig. 6(a). The grayed-out areas in Figure 6 mark table-size combinations that an optimizer would reject as sub-optimal, in favor of exchanging the build and probe tables to get better throughput.

5. CONCLUSIONS

In this paper we have shown how to efficiently offload large-scale relational join operations to the GPU, achieving a sustained overall throughput of 2 GB/s to 6.1 GB/s depending upon the size of the result set. This corresponds to 3–8× performance improvement over the fastest reported implementation on CPUs [10]. Adjusting for the current CPU generation still leaves us with more than half an order of magnitude speed-up. UVA allows us to achieve throughput rates close to the hardware capabilities, without requiring any hand tuning. Moreover, the input tables remain in host memory, obviating the need to manage data copies. In the typical scenario for a hash join, i.e., when the probe table is larger than the build table, our GPU hash join implementation was able to achieve 96 % utilization of the PCI-E generation 2 bandwidth (6.3 GB/s). PCI-E generation 3 doubles that bandwidth, and we expect the next generation GPUs to be able to take advantage of it. Future work includes support for handling larger data sets, e.g., spilling large hash tables to CPU memory and reading input tables from external storage, and processing more complex queries.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB'99*.

[2] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Ameta. *GPU Computing Gems: Jade Edition*, chapter 4, pages 39–53. Morgan Kaufmann, 2012.

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD'11*.

[4] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99*.

[5] R. Budruck, D. Anderson, and T. Shanley. *PCI Express System Architecture*. Addison-Wesley, 2003.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4).

[7] N. K. Govindaraju and D. Manocha. Efficient relational database management using graphics processors. In *DaMoN'05*.

[8] 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), Dec. 2009.

[9] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD'08*.

[10] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2), Aug. 2009.

[11] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowledge and Data Engineering*, 14.

[12] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS'11*.

Table 1: Measured Bandwidth to Device Memory on the GTX 580 (512 MB accessed by 1,024 blocks with 1,024 threads/block).

		Bandwidth GB/s		
		coalesced	non-coalesced	random
READ (data-dep.)	32 bit	122	4.8	3.8
	64 bit	151	10	7.7
READ (data-indep.)	32 bit	119	4.8	3.8
	64 bit	152	10	7.7
WRITE	32-bit	144	5.4	3.8
	64 bit	155	11	7.7
CAS	32-bit	8.6	2.4	2.3
	64 bit	11	4.9	4.6

APPENDIX

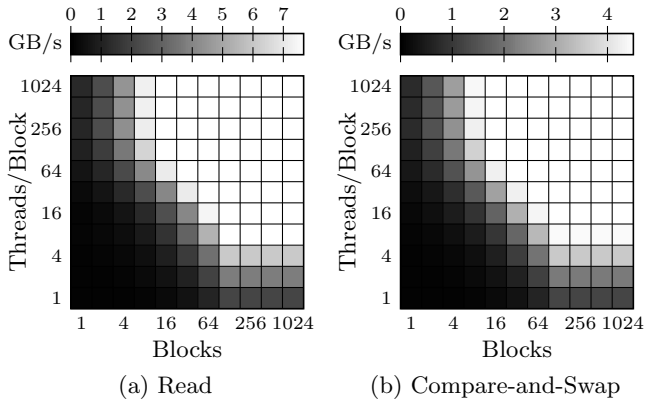
System Configuration. Throughout our experiments, we use a high-end consumer graphics card, a GeForce GTX 580. Its GF110 core is clocked at 1.54 GHz and encompasses 512 CUDA cores in 16 streaming multiprocessors with 32 CUDA cores each, and has access to 3 GB of GDDR5 device memory. The card was installed in a system with an Intel Core i7-2600 Sandy Bridge CPU clocked at 3.4 GHz and 16 GB of DDR3-1333 memory. We ran Suse Enterprise Linux 11.1 with a 2.6.32 kernel Nvidia graphics driver 285.33, and CUDA toolkit 4.1 installed. The system was dedicated to the experiments, i.e., with no other users or applications active. The video output on the GTX 580 was disabled and system's video output was handled by another graphics card.

A. DEVICE MEMORY ACCESS

Table 1 shows the measured bandwidth to the 3,072 MB device memory on the GTX 580 for different access patterns. The table reflects the well-known property of GPU memory; the effective bandwidth highly depends on the pattern and the type of memory access. In coalesced memory access, adjacent threads are accessing adjacent memory locations, e.g., thread t references memory location m, thread $t+1$ location $m+1$, etc. Non-coalesced accesses, e.g., where a single thread t references consecutive locations $m, m+1, \ldots, m+k-1$ should be avoided since they result in 15–25× lower memory throughput.

In this setting, we are considering read, write, as well as atomic *compare-and-swap* (CAS, `atomicCAS()`) operations on 32- and 64-bit data. Reads are further divided in data-dependent and data-independent accesses. In the former, the next memory location accessed depends on the content of the currently accessed location. Such a pattern is exhibited, for example, when traversing a linked list.

For our join implementation random access operations, that occur when accessing hash table entries, are of particular interest. Random CAS is used to coordinate the insert operations done concurrently by hundreds of threads during the create phase of the hash table. The random read happens during the probe phase of the hash join when the key and payload entries are looked up in the hash table. For 64-bit random accesses we note a peak bandwidth for reads of 7.7 GB/s and 4.6 GB/s for CAS.[8]



(a) Read (b) Compare-and-Swap

Figure 7: Measured bandwidth of 64-bit accesses to random locations within 512 MB of device memory on the GTX 580.

Figures 7(a) and 7(b) show the measured aggregate read and CAS bandwidth that can be obtained for different GPU thread configurations. In CUDA, a compute kernel running on the GPU is always executed in a grid of thread blocks. In particular, a thread block is mapped to one streaming multi-processor, and therefore, all threads of that blocks are executed by the CUDA cores on this multi-processor. The figures show that for both read and CAS a minimum number of thread blocks and threads/block is needed to achieve maximum performance. For reads and CAS at least 16 blocks have to be scheduled to achieve a peak bandwidth of 7.7 GB/s and 4.6 GB/s. This requirement is quite obvious; starting with 16 blocks all 16 multi-processor are used. Inside of a thread block we need at least 8 threads on this GPU to reach peak bandwidth for reads and CASs.

Similar to SMT in traditional CPUs, memory latency can be hidden by mapping more parallel activity on a core. In this case, while one thread (warp of threads) is waiting for data another thread (warp of threads) can be executed. Latency can be hidden (and throughput improved) when another thread is executed in the meantime. This interleaving can be applied all the way down to the actual DRAM access, where a column access in a bank can be interleaved with opening a new row in another currently idle bank.

The diagonals in Figures 7(a) and 7(b) correspond to a total of 1,024 threads. Hence, having accesses of at least 1,024 threads in-flight is necessary to keep all queues in the memory controller full. We cannot observe any degradation as we move towards the upper right corner in the figures by increasing the total number of threads. For this reason we can further increase the number of threads/block and block count in our join implementation as needed. In more complex GPU kernels than those used in these benchmarks, other resource constraints limit the number of blocks and threads/block that can be executed concurrently due to the register usage of a thread and the shared memory requirement of a block.

In Figure 8 we show the measured bandwidth for differ-

[8]For comparison to a CPU, the quad-core Sandy Bridge i7-2600 with DDR3-1333 memory in our experiments has an aggregate bandwidth of 700 MB/s for random 64-bit reads and 380 MB/s for 64-bit CAS using 8 threads.

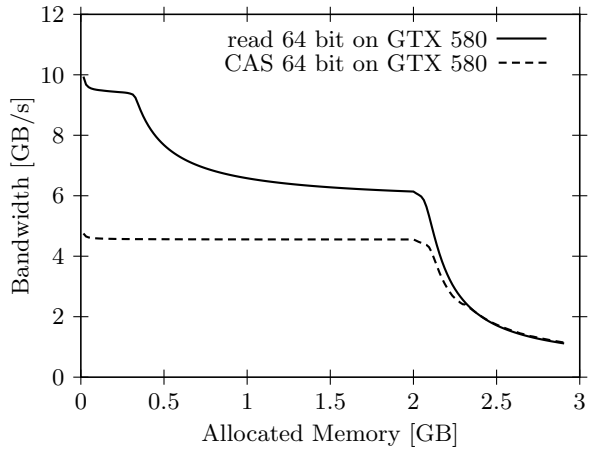


Figure 8: Measured bandwidth of accesses to random locations in device memory on the GTX 580.

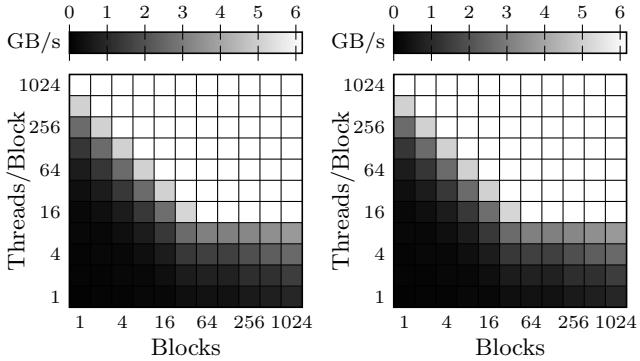


Figure 9: Measured bandwidth of 64-bit coalesced accesses to host memory using UVA for different CUDA grid and block configurations.

ent random access patterns for different amounts of allocated device memory. An interesting observation on our 3 GB GTX 580 is the bandwidth drop for random reads at 256 MB and a further sharp drop for all patterns when more than 2 GB/s are randomly accessed. We are unable to explain this behavior using publicly available data. However, we believe that the following provides an explanation for the performance drop in Figure 4 for table sizes ≥ 32 M tuple: If the GPU uses a common queue for both device memory and UVA accesses, the bandwidth drop for device memory accesses could slow down the UVA access and reduce the overall throughput even though the actual device memory bandwidth is still well above the PCI-E limit.

B. HOST MEMORY ACCESS

For the hash join algorithm UVA-based memory accesses is used in both the create and the probe phase to read the input tuples. The join results are written back to host memory via UVA as well. In both cases, memory accesses are coalesced. Figures 9(a) and 9(b) show the memory bandwidth for 64-bit read and write access to host memory through UVA for different thread configurations. We are able to obtain a sustained throughput of 6.17 GB/s for both access types. The figures show that in order to achieve peak bandwidth, at least 16 threads/block need to be used. This can be explained by the DMA transfer sizes supported by the GPU. The PCI-E endpoint of the GTX 580 supports a maximum payload size of 128 bytes in PCI-E Transaction Level Packets[9]. Using 16 threads, 16×64-bit requests translate into full 128 bytes payloads in the PCI-E packets. The diagonals (cf. Fig. 9) correspond to total of 1,024 threads that are required to achieve peak bandwidth. In other words, having accesses of at least 1,024 threads in-flight is required to keep the memory queues filled. Figures 9(a) and 9(b) show that we cannot observe any degradation as we move towards the upper right corner, and the total number of threads increases.

The measured bandwidth is close to the theoretical peak bandwidth for a 16-lane PCI-E generation 2 device as the following simple calculation shows: A single PCI-E generation 2 lane operates at 5 GT/s. Given the 8b/10b symbol encoding used in the PCI-E physical link, the nominal bandwidth thus is 500×10^6 B/s for a single lane and 8×10^9 B/s for all 16 lanes. With a 24 byte packet overhead,[10] the packet sent for a single 128-byte request is 152 bytes in size [5], and hence, the theoretical peak-bandwidth $8 \times 10^9 \text{ B/s} \times {}^{128}/_{152} \approx 6.27$ GB/s. The results in Figures 9(a) and 9(b) show that the measured bandwidth is 98 % of the theoretical peak value.

[9]The maximum payload size in a transaction level packet supported by a PCI-E device can be read from the *Device Capability Register*, for example, using `lspci -vv`.

[10]For a 24 byte packet overhead we assume that 64-bit host addresses are used without the optional end-to-end CRC field.