Ameliorating Memory Contention of OLAP operators on GPU Processors

Evangelia A. Sitaridi, Kenneth A. Ross *
Dept. of Computer Science, Columbia University
(eva,kar)@cs.columbia.edu

ABSTRACT

Implementations of database operators on GPU processors have shown dramatic performance improvement compared to multicore-CPU implementations. GPU threads can cooperate using shared memory, which is organized in interleaved banks and is fast only when threads read and modify addresses belonging to distinct memory banks. Therefore, data processing operators implemented on a GPU, in addition to contention caused by popular values, have to deal with a new performance limiting factor: thread serialization when accessing values belonging to the same bank.

Here, we define the problem of bank and value conflict optimization for data processing operators using the CUDA platform. To analyze the impact of these two factors on operator performance we use two database operations: foreign-key join and grouped aggregation. We suggest and evaluate techniques for optimizing the data arrangement offline by creating clones of values to reduce overall memory contention. Results indicate that columns used for writes, as grouping columns, need be optimized to fully exploit the maximum bandwidth of shared memory.

1. INTRODUCTION

GPU processors have been applied to various data management applications: scientific data management, OLAP and relational databases. The recent increase in GPU memories (e.g., 6GB per GPU for an Nvidia C2070 machine) makes GPU processors suitable for small to moderate size databases where the data working set is GPU resident. Such memory sizes are not so different from the CPU RAM capacities of just a few years ago that drove the development of main memory databases.

Another motivation for keeping the data resident in GPU memory comes from the increasing popularity of cloud computing. The Amazon Elastic Compute Cloud (EC2) allows one to reserve and use machines by the hour. The EC2 offers machines with 2 Nvidia M2050 GPUs for about $2.10 per hour (reservation price) or about one quarter of this rate (spot price).[1] Users do not have to provision local systems for peak loads. Instead, they adapt their computing requirements (and their costs) to their needs at different points in time. For example, at the end of a reporting period, an organization may reserve a large number of machines to do intensive data analysis. In such a context, users could reserve sufficiently many GPU machines to store their entire analysis data set in GPU RAM for a limited time, during which a large number of analytic queries would be run. The higher performance of GPU-based analytics makes this choice cost-effective, because shorter reservations would be needed than with CPU-based computations.

Due to the particular thread organization, complex memory hierarchy and high degree of parallelization of GPU processors, more effort is required to design database operators that reach maximum memory bandwidth. Efficient programming for GPU processors requires understanding of thread organization and different memory types. We focus on Nvidia's CUDA architecture, although other popular general purpose GPU architectures are similar in terms of thread-organization and memory-hierarchy. Our target platform is the Nvidia Tesla C2070, which has 14 Streaming Multiprocessors (SMs). There are 448 cores: each SM can process a warp of 32 threads simultaneously using a common instruction stream on different data.

GPUs have a radically different memory-hierarchy from a traditional CPU. Global memory is the largest type of memory but it has high latency: 400–600 cycles. Shared memory is used as a parallel, software controlled cache. Its size on the C2070 is 16KB or 48KB depending on the kernel configuration. The access time of each shared memory bank is 4-bytes per 2 cycles. To maximize performance, shared memory is organized into 32 banks, so that all threads in a warp can access different memory banks in parallel. However, if two threads in a warp access different items in the same memory bank, a *bank conflict* occurs, and accesses to this bank are serialized, potentially hurting performance.

The C2070 offers atomic operations on global and shared memory, where each thread that calls an atomic operation on a variable is promised that this variable will not be accessed by another thread until this operation is complete. Other threads trying to access the same address get serialized. This creates another possible form of contention be-

*Supported by NSF Grants IIS-1049898 and IIS-0915956, and by an equipment gift from Nvidia Corp.

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

[1]Prices taken from `http://aws.amazon.com/ec2/` in March 2012.

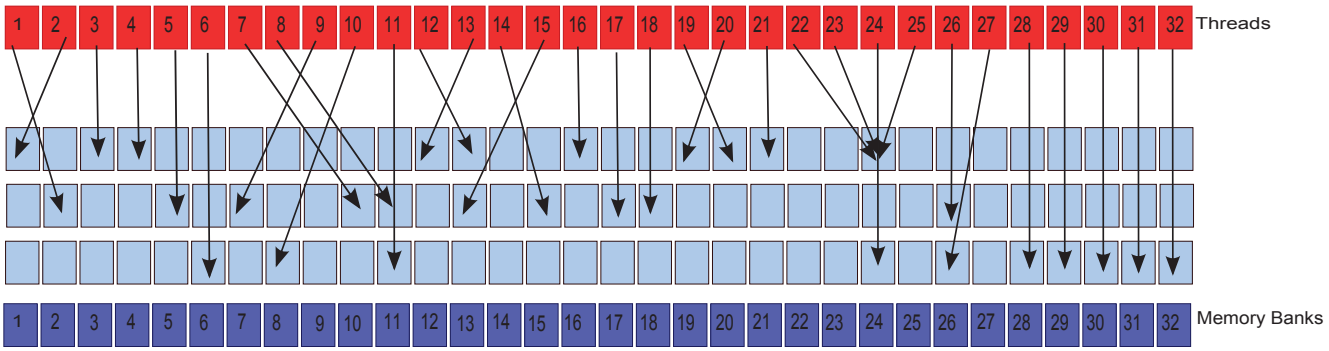

Figure 1: An example bank/value access pattern for 32 threads in a warp. Two serialization rounds are needed for reads, due to bank conflicts. Three serialization rounds resolve value conflicts for writes.

tween threads when at least one is writing data. We refer to this kind of serialization as a *value conflict*. Value conflicts can span warps, because atomic operations may still be in flight when new warps get scheduled to the SM.

Figure 1 illustrates a possible shared memory access pattern by the 32 threads in a warp. There are 2-way bank conflicts in banks 11, 13, 24, and 26. The three accesses to a single item by threads 22, 23 and 25 represent value conflicts. If these three accesses are reads, then there is no performance penalty; the system will broadcast the common data item to all three threads in one round. However, if these were write accesses, then two additional serialization rounds would be necessary for value conflicts. For additional discussion of the GPU architecture, see Appendix A.

Our goal in this paper is to examine the role of bank and value conflicts for database processing on GPUs, and to suggest bank optimizations for improving data access behavior. We focus on two core database operators: foreign-key join and grouped aggregation. For foreign key joins, shared memory is used to contain the needed fragment of the dimension table. For grouped aggregation, shared memory is used to contain the running aggregates for each group. In both cases, we make a scan through a fact table, consulting the structure in shared memory for each row.

Our key insight is that if there is additional shared memory available beyond that needed for the basic structure described above, we can use that memory to store extra copies of the underlying data. For foreign key joins, we store duplicate dimension table rows; for grouped aggregation, we store duplicate groups. In either case, we make sure that the duplicates and the original item all occupy different banks. We modify the base fact table so that some rows refer to one of the duplicates rather than the original item, in order to reduce the number of bank and value conflicts.[2] This modification is done once at data loading time, so the cost of optimization and the modification of the fact table is amortized over many queries.

Section 2 describes in more detail the problem of bank and value conflicts. Section 3 presents our solution resolving shared memory conflicts. Section 4 is an evaluation of our suggested techniques and Section 5 gives an overview of related work. Finally, in Section 6 we conclude and discuss future work.

[2]For grouped aggregation, a final pass combines the aggregates for the duplicates into a single aggregate for each item.

2. PROBLEM DESCRIPTION

We assume an in-memory OLAP setting and a star schema. Using coding techniques commonly used in OLAP databases [4, 23, 21], we assume that foreign key columns and grouping columns are coded with consecutive integer codes starting from 0. That way, dimension tables and aggregate structures can be organized as simple arrays rather than as hash tables.[3] There is a direct relationship between the array index and memory bank, since memory banks on the C2070 are distributed in a round robin fashion every four bytes. We assume all data tables are stored columnwise with 4-byte datatypes, maximizing the potential for bank parallelism.

If d is a foreign key column, then the domain of d in the initial fact table will be the integers between 0 and $c-1$ where c is the cardinality of the referenced table. If d is a grouping column, then the maximum value in the column is one less than the effective size of the aggregation table.[4] We will process a "warp's worth" of contiguous data at a time, a unit we shall call a "chunk." On the C2070, the chunk size is 32 data elements.

Now suppose that column d takes values 3 and 35 at two places in a single chunk of elements from column d. Because $3 \equiv 35 \mod 32$, both references will map to the same bank, leading to a bank conflict. For this example, we might create a new version of the element in slot 35, and put it in slot 3207, say, at the end of the table. In the fact table row with the conflict, we re-map 35 to 3207 and the conflict no longer holds because $3 \not\equiv 3207 \mod 32$. We keep track of this new row in slot 3207, which could be used for subsequent fact table rows as an alternative to slot 35 if slot 35 causes another conflict.

In the unbounded version of the problem, we do not limit the number of copies generated. If we're only concerned about bank conflicts within a warp, then 32 copies of each data item, one per bank, would guarantee that we could avoid bank conflicts altogether. In practice, fewer than 32 copies are needed. In the bounded version of the problem, we observe that the shared memory capacity puts a limit on how many values can be efficiently handled. Based on this

[3]Such optimizations are particularly important in GPUs. If different threads in a warp need to follow hash overflow chains of different length, then the execution paths will diverge and threads will be partially serialized for the length of this divergence.

[4]If some intermediate values don't appear at all in the table, then the actual grouping cardinality may be smaller.

Theta	Distinct Banks	Write SR	Read SR
0.00	20.42	3.54	3.44
0.25	20.41	3.53	3.42
0.50	20.36	3.57	3.38
0.75	19.94	3.81	3.18
1.0	18.24	5.33	2.77

Table 1: Average number of distinct banks, read serialization rounds and write serialization rounds in a chunk.

capacity, we set a budget on the average number of copies. For example, a budget of 5 would mean that the total size of the table including duplicates cannot exceed 5 times the size of the table without duplicates.

In the aggregation case, where we need to perform writes on the shared-memory-resident array, we also need to create copies to resolve value conflicts, where the same value appears more than once in a warp. We will also extend the analysis beyond the warp, looking for value conflicts between nearby warps within a fixed "window," on the grounds that an in-flight atomic update of a value might conflict with that same value in subsequent warps. In the worst case, more than 32 copies may be needed to completely avoid value conflicts.

Data partitioning between threads is static. Each thread in a thread block processes a certain number of records, so we know beforehand which records a thread is going to process. We can detect bank conflicts by scanning chunk-by-chunk and inter-warp value conflicts by remembering the set of values in the last chunks. The number of chunks we remember defines the optimization window. Our algorithm is easily extended for different regular access patterns, e.g., an access pattern where each thread reads four integers at a time using built-in CUDA vector types.

Static partitioning means that our optimization may not be effective if data items "move" from their original fact table grouping before being processed. This may limit our choices for other operators. For example, a selection operator that scanned the fact table and wrote an intermediate result containing only the matching records would change the chunking pattern. Alternative selection operators are compatible with retaining physical order. One option would be to combine the selection and aggregation into one joint kernel, so that the aggregation operator sees the data in the original locations. Another option would be to use a clustering scheme such as multidimensional clustering [19, 18] so that the records matching the selection conditions tend to be contiguous.

We consider a variety of Zipfian distributions for the fact table column, ranging from $\theta = 0$ (uniform) to $\theta = 2$ (very skewed). To give a better sense of the problem, we provide in Table 2 some statistics for a column of cardinality 1024 for different θ parameters. Without performing any optimization, we analyze the column and compute how many read and write serialization rounds are required, together with the number of distinct banks in a chunk. Note that skew hurts write serialization due to an increase in the number of value conflicts, but helps read serialization due to improved locality (since shared items can be broadcast to multiple threads).

3. ALGORITHMS

Before we discuss our main algorithms, we remark that it might be possible to reduce bank and value conflicts by reordering the fact table so that rows that would cause a conflict in the current chunk are held back until a later chunk. Reordering can only be a partial solution, because if a value occurs with a frequency higher than 1/32, then value conflicts cannot be eliminated by simple reordering. Further, there are often criteria more important than bank conflicts for ordering a fact table, so assuming the ability to reorder the table may be unreasonable.

3.1 Write Conflicts

Depending on the data distribution and data ordering each value should be assigned a different number of copies. Intuitively, frequently occurring values should get more copies, because those items are more likely to conflict, and because the extra copies are the most valuable when they can be used by many rows. Rather than statically choose a prioritization scheme for the number of copies based on frequency, we use a dynamic scheme to determine the number of copies for each value based on the "demand" for extra copies.

Initially each value is assigned one copy. We process a chunk of fact table rows at a time, until each chunk has been processed. For each chunk we proceed as follows.

We first try to assign as many values in the chunk as we can to one of its available copies, without causing any bank or value conflicts relative to previous choices. If we succeed at assigning all values, we move to the next chunk. If not, which is more likely, some values remain whose copies conflict with previous assignments. For each such value v, we assign v into one of the occupied banks and unassign the value v' that was previously there. We then try to reassign v' into one of its other copies, which could lead to a recursive sequence of reassignments. We do not consider reassignments at random. Instead, we use a breadth-first-search (BFS) algorithm to find the shortest sequence of reassignments that allows the value to be inserted without conflicts.

Assignment of a value in a chunk fails for one of the following reasons: 1) the distinct number of banks among all copies in the chunk is less than the number of banks, or 2) none of the keys can be assigned to an empty bank, because an empty bank is not reachable given the current set of copies. In both cases, to resolve the conflict a new copy of the value is created in one of the empty banks. In this way, we generate new copies of the values that are hardest to place. If we have already spent our space budget, we place the item without creating a new copy, and accept that this chunk will need multiple serialization rounds.

Two important choices affecting the space and time efficiency of the algorithm are:

- When failing, multiple bank-slots might be available. We want to assign the same number of copies to each bank-slot so that the replicated table is stored contiguously in the memory without gaps. If more copies are assigned to certain banks then there will be "holes" in the memory in the less popular banks. These holes still consume shared memory, and should be avoided.

- The order according to which we insert the copies into the BFS queue matters. If we always enqueue the lower numbered banks first, then there is a high probability that the available slots upon failure will be the higher

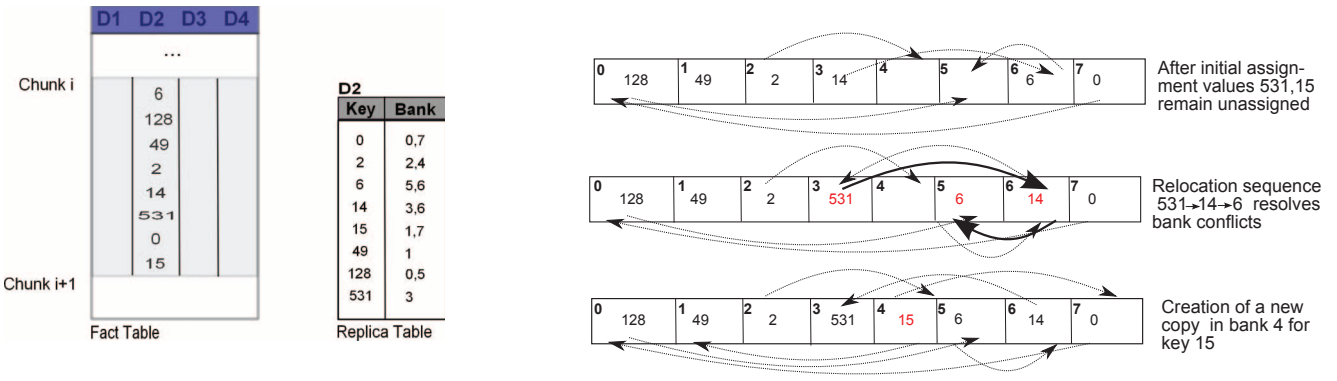

Figure 2: In this example we assume chunks of 8 elements and an equal number of shared memory banks and we show how our algorithm optimizes the second fact table column for bank conflicts. After the initial assignment, values 531 and 15 remain unplaced. A valid relocation sequence is found for value 531 (shown in bold edges). For value 15 there is no valid relocation sequence so a new copy is created in bank 4.

numbered slots, creating contention on those banks. To address this problem, we start each chunk from a different position enumeration of the copies.

Figure 2 shows how our algorithm processes a chunk of the second column of a fact table to eliminate bank conflicts. For simplicity, the chunk size in the example is 8 elements, equal to the number of threads in a warp and equal to the number of memory bank-slots. Each dashed edge links a placed value to its alternative bank locations. After the initial assignment, values 15 and 531 remain unassigned. In the next step, a relocation sequence is found for 531 that displaces 14, displacing 6 in turn. For value 15 there is no valid sequence of value movements because the only empty bank-slot 4 is unreachable, so a new copy of 15 is created in bank 4.

3.2 Read Bank Conflicts

In case of read conflicts the problem is relaxed due to the value multicasting performed in the hardware. If in a chunk there are some duplicate values, then all of those values can be assigned to the same bank without degrading the performance. This means that we can simply run the assignment algorithm for the first occurrence of the value in the chunk and put the rest of the occurrences in the same bank.

3.3 Inter-Warp Value Conflicts

Inter-warp value conflicts occur only between warps belonging to the same thread-block. We set a window size corresponding to the number of chunks prior to the current chunk to consider for value conflicts. (A window size of zero means that inter-warp value conflicts are ignored.) We shall examine the impact of window size experimentally. We extend the BFS algorithm so that copies that have previously been used within the current window are not used again in the current chunk.

In case of failure, we create a new copy as before. If we have already used the space budget we try again to place the value in the current chunk, ignoring inter-warp conflicts.

Finally, for a skewed dataset with a large window the number of copies for frequent values will also be high, increasing the search cost. To reduce this cost we consider first the copies that were least recently used, increasing the probability that a non-conflicting assignment is found early.

4. EXPERIMENTAL RESULTS

For our experiments we used synthetic data following the zipf distribution for different θ parameters. The default number of distinct values in the zipf distribution was 1024. Each column is a 4-byte integer. We used our suggested technique to resolve bank and value conflicts for different table sizes (t, up to 200M which is the maximum number fitting in GPU memory), window sizes (w) and space budgets (b).

We ran the following queries on an OLAP star-schema:

```
Q1: SELECT SUM(D1.B)          Q2: SELECT A, COUNT(*)
    FROM F, D1                    FROM F
    WHERE F.A=D1.A                GROUP BY A
```

In both cases fact table `F` was stored in the global memory. In the first query dimension tables are stored in the shared memory to perform the foreign key join, and a scalar aggregate is generated. In the second query shared memory is used to store the aggregates local to a thread block. After each thread-block computes the sums in the shared memory, it merges the results for each copy to global memory; in the end, global memory contains the correct aggregate sums. In what follows, results measuring read conflicts correspond to `Q1`, and results measuring write conflicts correspond to `Q2`.

Optimization was done on a dual-chip Intel E5620 CPU using 16 threads. GPU performance was measured on an Nvidia Tesla C2070 machine with 6GB of RAM and a nominal RAM bandwidth of 144GB/s. The GPU was configured to use 48KB of shared memory in each SM. Each thread-block processed 350K rows using 1024 threads. The number of thread blocks for a kernel was computed based on the number of table records.

Figure 3 shows the number of copies per value for different optimizations with a very lenient space budget b of just under 12 copies. Writes generate more copies than reads, because the write value conflicts create additional constraints. For similar reasons, the number of copies increases as the window size is increased. As skew increases, the number of copies decreases significantly. Many copies of a few popular items is often enough to create a conflict-free access pattern. As the number of records increases, the number of copies also increases, but the increase is fairly mild after 50M records. Figure 4 shows the average number of copies

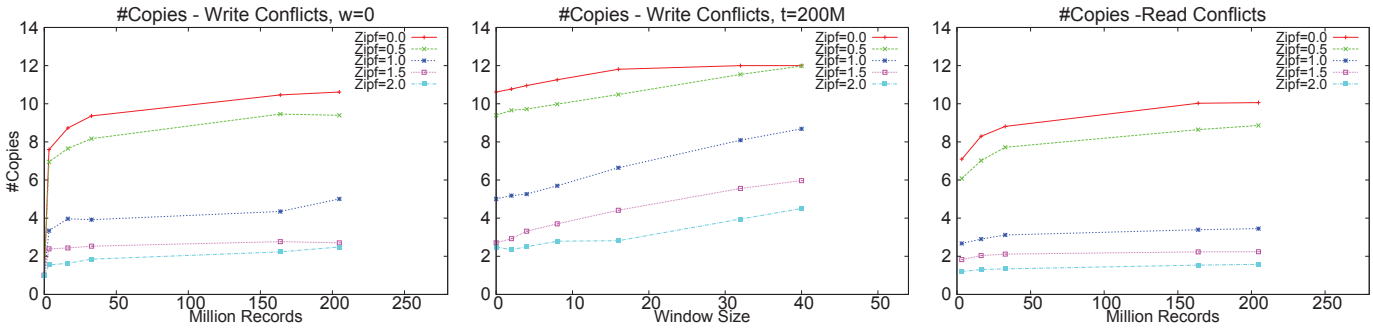

Figure 3: Number of copies per value for different table sizes, and different θ parameters

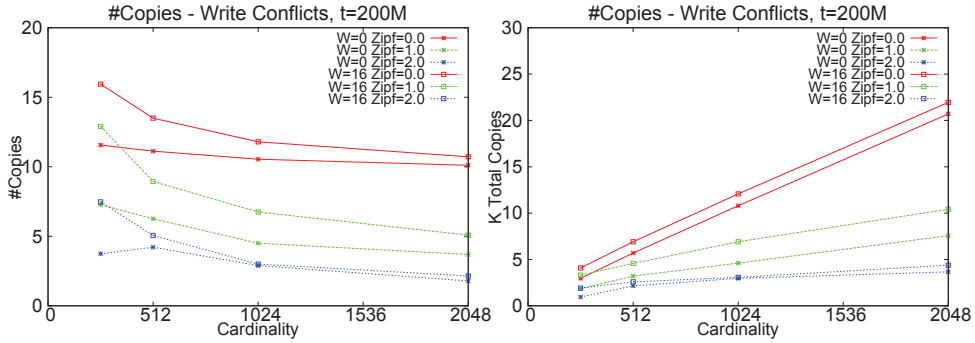

Figure 4: Number of copies per value for varying cardinality

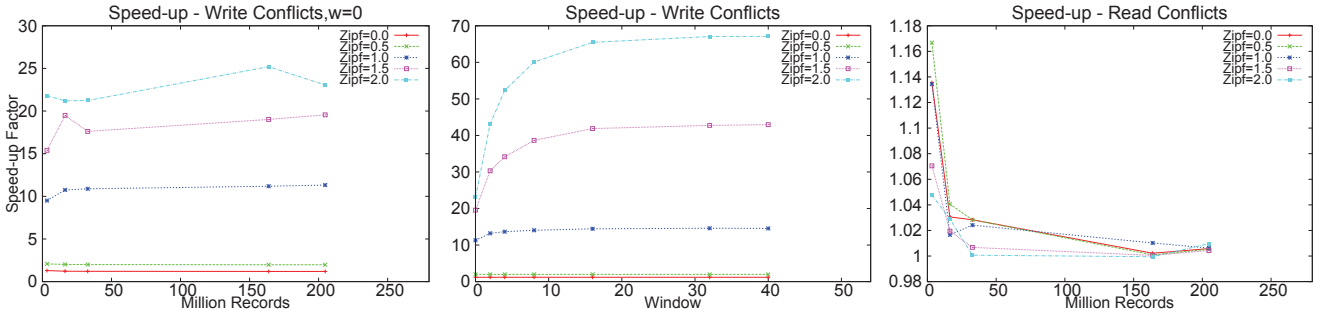

Figure 5: Speed-up for Read and Write Conflicts

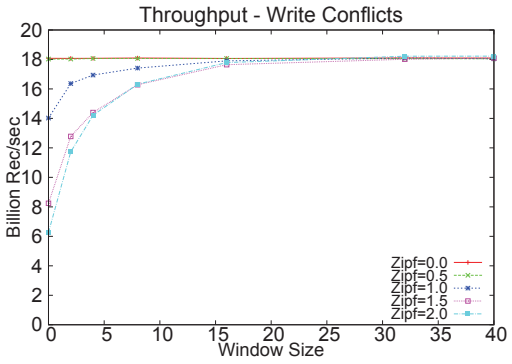

Figure 6: Throughput for different windows

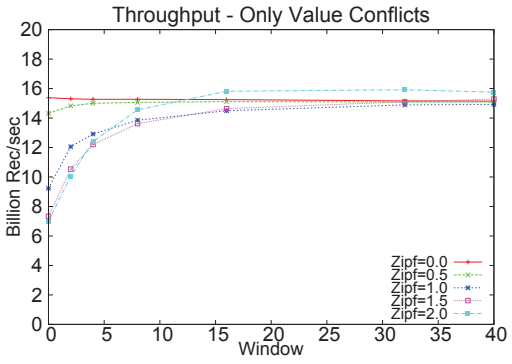

Figure 7: Optimizing for value conflicts only

and total number of replicated values for varying column cardinality. As expected, for lower cardinalities the number of copies is higher because there is an also higher probability for value conflicts and this trend is more apparent when an optimization window is set.

Figure 5 shows the speed-up of query execution on the GPU for the configurations of Figure 3. The write speed-ups are particularly dramatic at high skew, highlighting the importance of addressing conflicts when there are heavy hitters. For uniform data, the speed-up factor is about 1.2, showing that optimizing for write conflicts is still important without heavy hitters. The window size is unimportant for uniform data, but is important for skewed data. Most of the benefit of windowing occurs with a window size of 16 chunks. For reads, the speed-up is much smaller, about 2% or less once there are enough records so that thread scheduling can hide the read serialization latency. As previously noted, unlike for writes, skew helps reads because it provides more opportunities for values to be broadcast to multiple threads. In Appendix A.3 we describe a worst-case scenario for read conflicts where all threads in a warp read a different value on the same bank. We note that since read conflicts are just a sub-case of write conflicts, columns that are both read and written by various queries should be optimized for writes.

In Figure 6 we show the actual throughput for different windows, where the table size is 200M records. We can process about 18 billion records per second, i.e., about 72GB/sec. To assess the importance of optimizing writes for bank conflicts, we repeated the experiment with a modified algorithm that optimizes for value conflicts but not bank

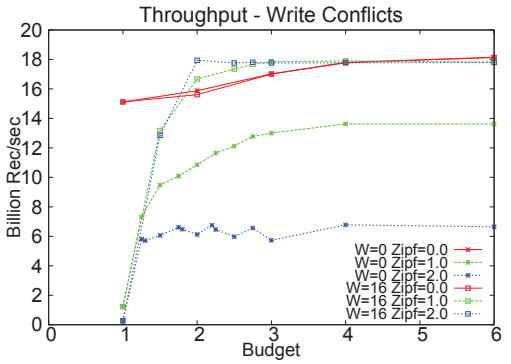

Figure 8: Throughput for different budgets

conflicts. The results in Figure 7 show that there is a 20% drop in throughput relative to Figure 6. Write bank conflicts appear to be more significant than read bank conflicts. Atomic write operations take longer, since they need a read-modify-write cycle.

For $\theta = 2.0$ in Figure 7, the performance is better than expected. At this extreme level of skew, there are two heavy hitters that occur many times in each chunk. Both of these heavy hitters have copies in every bank, so they are easy to place. The only possible bank conflicts come from the less frequent items, of which there are just a few in each chunk. The expected number of bank conflicts resembles the birthday paradox: The number of people in a group of size n having the same birthday as another member is proportional to n^2. By removing a subset of items this expectation also decreases quadratically. Thus high-skew distributions indirectly optimize for bank conflicts, even when only value conflicts are explicitly considered by the placement algorithm.

Figure 8 shows how performance depends on the space budget. For uniform data, we get close to maximum throughput at an average of 4 copies per value. For skewed data, even 2 copies per value gives good performance: our algorithm first creates copies for the frequent items that cause most of the conflicts. These results show that with a realistic (2–4X) increase in shared memory footprint, one can get most of the benefits of bank and value conflict avoidance.

In Figure 9 we see the time performance of the optimization algorithm for write conflicts. The elapsed time is just a few seconds, even for moderately large window sizes. For increasing skew, the algorithm runs faster because it is easier to arrange the records in a chunk. Our algorithm adjusts to the data distribution by creating many copies for the frequent values, so we have the freedom to place them in any bank and only have to resolve conflicts with the non-frequent values. However, for increasing window sizes, skewed data needs longer optimization time because relocations of frequent values are expensive, due to the high number of copies these items have. Reassignments of frequent items occur when infrequent values with few copies have to displace them.

5. RELATED WORK

There has been prior-work on database processing using GPU processors. Conjunctive selections and aggregations were accelerated on earlier GPU processors with a different memory architecture [9]. A subset of SQLite commands has

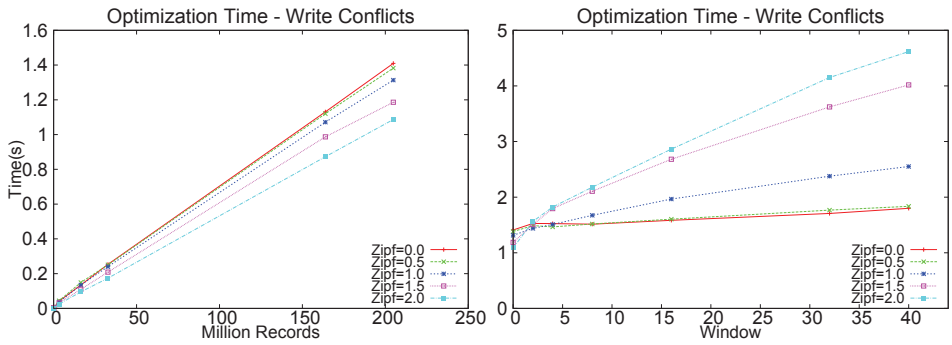

Figure 9: Optimization Time

been implemented on GPU processors resulting in significant speed-ups [1]. The power of GPU processors has also been exploited for the efficient implementation of different join algorithms [10]. Scatter and gather operations have been optimized for GPU processors to improve data locality [13]. Radix-sort was implemented with these scatter/gather operations while using shared memory to improve performance. A Map-Reduce framework has been suggested to facilitate programming of web analysis tasks on GPU processors without sacrificing performance [12].

An extended precision library has been implemented on GPUs and incorporated into a GPU-based query engine to achieve a significant performance improvement for scientific applications [16]. The bottleneck of transferring the data from CPU to GPU has been studied, optimizing the selection between GPU and CPU, suggesting the GPU as a co-processor [11, 7]. Alternative database compression algorithms have been employed to alleviate the transfer bottleneck [8]. FAST, an architecture sensitive tree index suitable for CPU and GPU processors been suggested to accelerate in-memory search [15].

Alternative aggregation strategies on chip multiprocessors have been studied to minimize thread-level contention on CPUs exhibiting different degrees of memory sharing between threads [2]. A framework for parallel data-intensive operations automatically detects and responds to contention by cloning popular items at query time [3]. This and two additional parallel aggregation strategies have been studied on a Nehalem processor [24].

Cuckoo hashing methods resolve hash collisions using multiple hash functions for each item instead of one [20, 6, 22]. During the insertion of an item, if none of the positions are vacant, the key is inserted in one of the candidate positions, selected randomly, displacing the key previously placed there. The displaced key is re-inserted in one of its alternate positions. This procedure is repeated until a vacant position is found or a maximum number of re-insertions is reached. Our method searches for the shortest relocation sequence that eliminates bank conflicts, instead of following a randomized procedure.

Data declustering techniques are used to distribute data partitions among multiple storage units, e.g., disks [14] or servers. Replication and optimal replica placement of data items has been suggested to maximize resource utilization in the Kinesis distributed storage system [17].

6. CONCLUSIONS

We defined the problem of bank conflicts and value conflicts for data-processing operators on GPU processors. We studied the impact on performance of those two contention factors for two popular OLAP operators on CUDA architecture. We suggested and evaluated a technique for resolving conflicts that can easily be configured for different memory access patterns and space budget requirements. Results indicate that columns that are written by various queries e.g., potential grouping columns, should be optimized for writes and that read conflicts should not be a high priority for bank optimization. We plan to apply the same technique on clustered tables using separate structures for fragments of the fact table to take advantage of local skew. We also plan to extend our technique for different key sizes (e.g., 2-byte and 8-byte), given that GPU processors favor 4-byte accesses.

7. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU*, 2010.

[2] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[3] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, 2010.

[4] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Conference*, 1985.

[5] N. Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, November 2011.

[6] U. Erlingsson et al. A cool and practical alternative to traditonal hash tables. In *Workshop on Distributed Data and Structures*, 2006.

[7] R. Fang et al. GPUQP: query co-processing using graphics processors. In *SIGMOD*, 2007.

[8] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3, 2010.

[9] N. K. Govindaraju et al. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[10] B. He et al. Relational joins on graphics processors. In *SIGMOD*, 2008.

[11] B. He et al. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34, 2009.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT*, 2008.

[13] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Supercomputing*, 2007.
[14] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *SIGPLAN Not.*, 1992.
[15] C. Kim et al. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.
[16] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *DaMoN*, 2010.
[17] J. MacCormick et al. Kinesis: A new approach to replica placement in distributed storage systems. *Trans. Storage*, 2009.
[18] V. Markl, F. Ramsak, and R. Bayer. Improving olap performance by multidimensional hierarchical clustering. In *IDEAS*, 1999.
[19] S. Padmanabhan et al. Multi-dimensional clustering: A new data layout scheme in DB2. In *SIGMOD*, 2003.
[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51, 2004.
[21] P. Pucheral, J.-M. Thevenin, and P. Valduriez. Efficient main memory data management using the DBGraph storage model. In *VLDB*, 1990.
[22] K. A. Ross. Efficient hash probes on modern processors. In *In Proceedings of the 23nd International Conference on Data Engineering*, 2007.
[23] K.-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus system. *ACM TODS*, 15(1), 1990.
[24] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

APPENDIX

A. CUDA ARCHITECTURE

A.1 Thread Hierarchy

All threads belonging to a thread block are executed in the same SM, and each block is executed independently from other blocks. A block has a maximum number of threads, which in current high-end processors can be up to 1536 threads. A thread-block is grouped into a set of warps, e.g., a block of 1536 threads has forty-eight 32-thread warps.

CUDA employs a Single Instruction Multiple Thread (SIMT) architecture where threads in a warp start at the same program address, but keep private register state to execute on independent data. Performance is maximized when all threads in a warp follow the same execution path. When they diverge, such as during a conditional branch, their execution is serialized.

A.2 Memory Hierarchy

To fully utilize the memory bandwidth of a GPU programmers should carefully design their memory access patterns. CUDA memory spaces are either located off-chip or on-chip. Off-chip memories are more plentiful but have higher access latency. *Global* and *local* memory are located off-chip, while *shared memory* and *registers* are located on-chip.

Off-chip Memories

All threads access the same global memory space. Global memory accesses should be *coalesced* within a warp to minimize memory transactions. The simplest example of an optimal coalesced memory access is when all threads in a warp access consecutive 4-byte addresses. Local memory is local to a thread and has similar latency to global memory.

On-chip Memories

Shared memory is available to all threads in a thread block for shared access. On the Nvidia C2070, its size is 16KB or 48KB per SM, depending on the kernel configuration. Shared memory uses the same circuits as the L1 cache, and a programmer can configure how much memory is allocated for shared memory space and how much for the L1 cache, depending on the memory access pattern of the kernel. Shared memory can be both read and written by the threads in a block.

Shared memory is divided into banks and their number is equal to the number of threads in a warp. Banks are interleaved so that consecutive 4-byte words belong to different banks. If all the memory requests of the threads in a warp fall into different banks, they can be serviced in parallel. However, if two or more threads access the same bank for different items, the access to this bank are serialized. If there are multiple conflicts in a memory request, the hardware splits this request into as many conflict-free requests as are needed. A multicast mechanism is implemented for read requests where a 4-byte word being read by multiple threads in a warp can be broadcast to all requesting threads simultaneously. A read bank conflict thus occurs only if multiple threads accessing the same bank request a *different* 4-byte word. If the keys are 8-bytes the memory request of a warp is split in two requests, one request per half-warp [5]. As a consequence, for a read-request there is a bank conflict if two or more threads in either of the half-warps access different addresses of the same bank.

A register's scope is a single thread. The access latency of a register is 0 cycles, assuming there are no read-after-write dependencies. Register memory only stores static arrays, and high register-usage limits the parallelism by limiting the number of threads in a thread block, due to the limited size of the register file.

Atomics

CUDA offers atomic arithmetic and bitwise functions on global and shared memory. Shared memory atomics are significantly faster than global atomics. Atomic functions perform read-modify-write operations on 32 or 64-bit words: when a thread performs an atomic function it is guaranteed that no other thread will interfere until this operation is finished. When multiple threads write atomically to the same address the accesses to this address are serialized.

A.3 Impact of bank and value conflicts

We used the CUDA command line profiler to count the number of bank conflicts for different degrees of conflicts. The CUDA profiler reports counters per SM. We profiled the performance of the same operators as in Section 4. We generated synthetic data causing a specified number of serialization rounds per warp. We were careful to make sure that all values are distinct within a chunk, to show the worst case scenario for read conflicts. Figure 10 shows the throughput and the number of conflicts as reported by the `l1_shared_bank_conflicts` counter of the CUDA Profiler. For the worst case of read conflicts the throughput is less

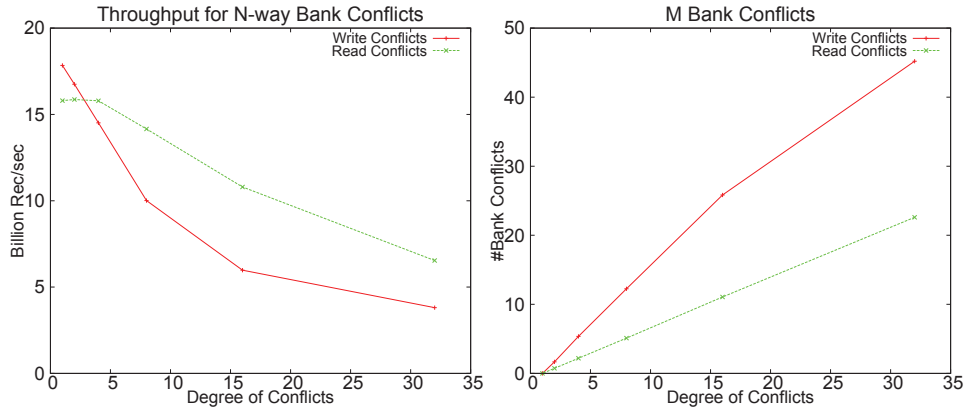

Figure 10: Throughput and Profiler Counters for varying conflict degree, t=30M

than half of the conflict-free performance. As previously noted, the effect for write conflicts was more significant.

One may wonder whether an access pattern that reads many different elements concentrated in a few banks is realistic. After all, a simple randomization of the bank location would lead to a reasonable spread of items across banks. The following example suggests that degenerate cases may indeed arise in practice.

Consider a foreign key join where the foreign key consists of two attributes, x, and y. Imagine that the dimension table represents metadata about cells in an n by m grid, so there are nm rows in total. Suppose that the dimension table is clustered by (x, y). The fact table also contains attributes x and y, but the fact table is clustered by y. As we scan through the fact table we will be repeatedly (for each y value) touching n dimension table rows separated by m rows. If d is the greatest common divisor of m and 32, then this access pattern will use only $32/d$ banks. In the worst case, m is a multiple of 32, and only one bank is accessed.