2015 International Conference on Parallel Architecture and Compilation

Stadium Hashing: Scalable and Flexible Hashing on GPUs

Farzad Khorasani Mehmet E. Belviranli Rajiv Gupta Laxmi N. Bhuyan
Computer Science and Engineering Department
University of California Riverside, CA, USA
{fkhor001, belviram, gupta, bhuyan}@cs.ucr.edu

***Abstract*—Hashing is one of the most fundamental operations that provides a means for a program to obtain fast access to large amounts of data. Despite the emergence of GPUs as many-threaded general purpose processors, high performance parallel data hashing solutions for GPUs are yet to receive adequate attention. Existing hashing solutions for GPUs not only impose restrictions (e.g., inability to concurrently execute insertion and retrieval operations, limitation on the size of key-value data pairs) that limit their applicability, their performance does not scale to large hash tables that must be kept *out-of-core* in the host memory.**

In this paper we present *Stadium Hashing* (Stash) that is scalable to large hash tables and practical as it does not impose the aforementioned restrictions. To support large out-of-core hash tables, *Stash* uses a compact data structure named *ticket-board* that is separate from hash table buckets and is held inside GPU global memory. Ticket-board locally resolves significant portion of insertion and lookup operations and hence, by reducing accesses to the host memory, it accelerates the execution of these operations. Split design of the ticket-board also enables arbitrarily large keys and values. Unlike existing methods, Stash naturally supports concurrent insertions and retrievals due to its use of *double hashing* as the collision resolution strategy. Furthermore, we propose *Stash* with *collaborative lanes* (clStash) that enhances GPU's SIMD resource utilization for batched insertions during hash table creation. For concurrent insertion and retrieval streams, Stadium hashing can be up to 2 and 3 times faster than GPU Cuckoo hashing for in-core and out-of-core tables respectively.

***Keywords*-GPU hashing; concurrency support; Out-of-core hash tables; key-value pairs; collaborative lanes execution;**

I. Introduction

In recent years, general purpose computing on GPUs has undoubtedly attracted much attention due to GPU's ability to achieve high performance on data-intensive applications. Although efficient GPU implementations of many prominent algorithms and data structures have been explored, hash tables for storing key-value pairs have received very little consideration. Existing hashing solutions on GPUs were generally designed for, and inherited from, the GPU's Graphics-only era while recent advances in GPU architectures have brought many more applications into the realm of GPU computing that require fast and flexible hashing solutions. Important applications such as dictionary based data compression [25], detection and elimination of duplicate objects in graph processing [15], and text mining [23] use hash tables; therefore they can benefit from GPU's massively multithreaded environment due to their data-intensive nature.

A common approach to exploiting parallelism for hashing in a multithreaded environment is to simultaneously perform multiple insertions or multiple retrievals of key-value pairs. When insertions are performed in parallel, to correctly handle collisions, a strategy known as *chaining* is employed – techniques for GPUs that use this strategy include [1], [2], [7]. Chaining atomically swaps to-be-hashed key with the key that is already present in the bucket; the swapped out key is *rehashed* and this process of swapping and rehashing is repeated until an empty bucket is found. Nonetheless, employing chaining leads to a number of restrictions. First, chaining based methods restrict concurrency to operations of the same type – insertions and retrievals cannot be performed at the same time. Workarounds to enable safe concurrent mixed operations inevitably degrade performance. Second, they are primarily designed for the scenario where the hash table can be held in GPU's global memory thus limiting their effectiveness for large scale data processing. Third, when some of the parallel operations by threads in the same SIMD group are unsuccessful, all the threads in the group must wait for their completion. This imbalance leads to degradation in GPU's SIMD resource utilization. Finally, since they use *atomic exchange* to deploy *chaining*, the maximum supported size of atomic operation restricts the size of key-value pairs (currently 64-bits).

In this paper we present *Stadium Hashing* (Stash), a parallel hashing method that utilizes a compact auxiliary data structure called *ticket-board*. For every bucket in the table, the ticket-board contains a *ticket* consisting of a single *availability* bit and possibly multiple *info* bits. The availability bit indicates whether the bucket is occupied or not – threads can unset (zero) the bits atomically to safely reserve a bucket. *Stash* overcomes the restrictions associated with current hashing methods as follows. Stash employs *double hashing* as the collision resolution technique for quick empty bucket discovery. As a result, unlike existing methods that employ chaining, *Stash* allows concurrent insertions and retrievals and also load factors as high as one. *Stash* achieves scalability by keeping the large hash table in the host memory while holding the *ticket-board* inside the GPU's global memory. This scheme enables fast resolution of the threads' transactions using per-bucket tickets. The info bits are derived from the key that resides in the bucket; thus, they can help to quickly reject a table bucket without having to access the table bucket content itself hence speeding up lookups. In addition, the separation between the table and the ticket-board allows us to have arbitrarily large keys and values; thus overcoming another restriction of the existing methods.

Furthermore, we propose stadium hashing with *collaborative lanes* (*clStash*) that increases the SIMD resource utilization of batched insertions. When multiple threads are concurrently asked to perform insertions, some threads might be successful in their first attempt while other threads in the same SIMD group may not succeed. In clStash, instead of making successful threads wait, unsuccessful threads accumulate unfinished requests in shared memory and thus all threads in the SIMD group can continue by fetching fresh requests from GPU's global memory. When enough unfinished requests have been accumulated in the shared memory, successful threads fetch them while unsuccessful threads retry working on the requests they had from the previous iteration. This strategy keeps all the threads inside a SIMD group busy. Finally, we propose a hybrid of Stash and clStash to improve insertion performance.

The key contributions of this paper are:

- We propose *Stash*, a parallel hashing scheme that provides efficient concurrent execution of mixed operation types, delivers excellent out-of-core performance via use of an auxiliary compact data structure named ticket-board, and allows having large keys and values;
- We present an efficient implementation of Stadium hashing in CUDA framework, and show that it can provide up to 2x and 3x speedup against GPU Cuckoo hashing over concurrent in-core and out-of-core inputs respectively; and
- We introduce *clStash*, a solution for *batched insertions* that greatly improves utilization of GPU SIMT architecture. *clStash* warp execution efficiency is around 2x and 3.5x higher than *Stash*'s and Cuckoo-GPU's respectively.

In the rest of the paper, we review the limitations of the most prominent existing GPU hashing method in Section II. Section III describes Stadium Hashing and elaborates upon its efficient implementation in the CUDA framework. Section IV presents experimental evaluation. Sections V and VI give related work and conclusions.

II. Motivation: Limitations of Related Work

The most notable endeavor to implement a hash table on the GPU and perform either parallel insertions or parallel retrievals is by Alcantra et al. [2], which is an improvement over the same authors' previous work [1]. Alcantra et al. use the Cuckoo Hashing [18] method and rely upon atomic exchange operations to safely use multiple threads during hash table construction.

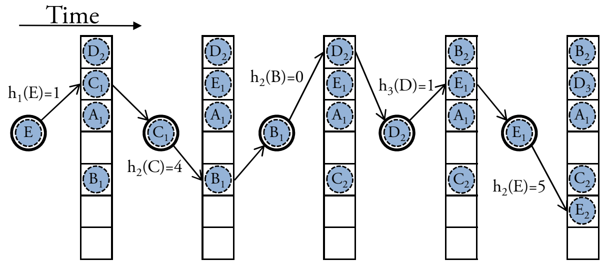


Figure 1. Insertion in a cuckoo hash table is a recursive procedure. Insertion of a *pair* can cause a long eviction chain.

Insertion: Cuckoo hashing uses multiple hash functions to provide a key with multiple insertion locations. For a given key, the first hash function gives the bucket address for the key-value pair to reside. The key-value pair that is already residing at that address is swapped with the new key-value pair. To reinsert the swapped out key-value pair, another hash function, different from what hashed the pair into their previous location, is used to find a new bucket. This procedure is repeated until an empty location is found. Figure 1 shows this recursive insertion procedure.

[2] performs the insertion and eviction of key-value pairs using 64-bit atomic exchange operations so that multiple threads can safely update the hash table. Every thread participates in an insertion chain until an empty slot is found. Generally, Cuckoo hashing needs to know which hash function placed evicted key into its last position so that it can select the next hash function, in a round robin fashion, to attempt reinsertion. In [2] the previously used hash function is found as follows. All the hash functions are applied to the evicted pair, and by comparing the generated addresses with the address of the bucket from which the key-value pair was evicted, the appropriate hash function is identified.

Retrieval: When the value for a key is queried, cuckoo hashing performs hash functions one by one on the key. It then checks the content of table addresses and once a matching key is found, the corresponding value is returned.

Limitations: Cuckoo hashing on GPU [2] suffers from a number of issues and introduces restrictions via its design:

- *Restriction on concurrency.* It does not support concurrent insertion and retrieval. A lookup for a query key may fail simply because during the probe the already-inserted pair is in transit, i.e. another thread being in an insertion-eviction chain is currently holding the pair containing the query key in its registers. For example in Figure 1, querying C in the second step will fail although it had been inserted previously. We modified GPU Cuckoo hashing to accept mixed insertions and retrievals while not processing them concurrently. We compared the performance of Cuckoo hashing for mixed operations with the case when all insertions are performed before lookups using randomly generated key value pairs.

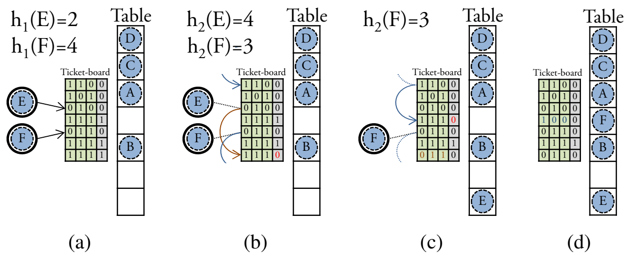


Figure 2. An example - Inserting two pairs in a table with 7 buckets & ticket size 4; the procedure starts from (*a*) and ends with (*d*); steps created by double hashing ensure traversal of all entries.

- *Performance degradation for large table sizes.* Since [2] involves atomic exchange operations that *ask for atomics returned values*, when the table is large and thus resides in the host memory, the latency associated with reads during insertions and possibly multiple reads during retrievals introduces long waiting times that degrade performance. In other words, further instructions cannot be scheduled for the thread until the content of the host memory position has traveled over the slow PCIe bus and arrived at the device. Long eviction chains make this problem even worse.
- *Inefficient use of SIMT hardware.* In [2], failure in the insertion of even a single key-value pair into the table by one thread causes starvation of all other SIMD threads (warp lanes, in CUDA terms) that have successfully inserted their pairs. By profiling [2] we have observed insertions exhibit low warp execution efficiency of around 25%.

Moreover, [2] restricts the key and the value to occupy 32 bits and can also fail when the Load Factor is high. In summary, Cuckoo hashing on GPU [2] has restrictions that limit its practicality and performance. This motivates the need to explore practical hashing approaches that are flexible (i.e., more general), scalable, and provide high utilization of GPU's SIMT hardware.

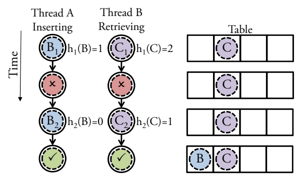
III. Stadium Hashing

In this section, we first introduce Stadium Hashing (*Stash*) that addresses the first two issues with Cuckoo Hashing [2]: it allows concurrent execution of mixed operation types, and efficiently scales to large table sizes. We also show efficient implementation of Stash in the CUDA framework. Then we present Stash with collaborative lanes (*clStash*), that improves the warp execution efficiency of the underlying SIMT architecture during insertions.

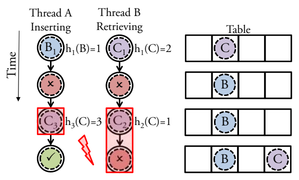
Stash Data Structures: Stadium hashing uses a split design with two structures: a *table* for key-value pairs, of any size, that can be kept in either GPU global memory or the host memory depending upon its size; and a compact auxiliary structure called *ticket-board*, that is kept in GPU global memory. The *ticket-board* achieves acceleration by maintaining a *ticket* corresponding to every bucket in the *table*. A *ticket* consists of a single *availability* bit and a small number of optional *info* bits. The *availability* bit indicates the occupancy status of the table bucket – a 1 bit indicates the bucket is empty and a 0 bit means it has already been reserved or occupied. In each *ticket*, if the corresponding bucket is occupied, a number of *info* bits, that have been generated deterministically from the *key*, provide a clue about the existing *key* in the bucket. *Info* bits are advantageous for prompt rejection of wrong table positions during retrievals without having to access the bucket content. While a *ticket* with the size of 1 bit does not have any *info* bits, a *ticket* with the size 8 carries 7 ticket *info* bits – in experiments, we consider ticket sizes of 1, 2, 4, and 8 bits. Let us see how the insertion and lookup operations are performed in Stash.

Insertion: To insert key-value pairs into the table, we assign each pair to a separate GPU thread. After hashing the key, the thread tries to book a bucket in the table by atomically unsetting (zeroing) the corresponding availability bit. If the availability bit in the returned value is 1, the ticket had been available and has now been acquired by the thread. Thus, the thread can store the info bits (in case the ticket size is not one) and insert the key-value pair into the table. Otherwise, the bucket had already been in use; hence the thread must rehash. *Stash* resolves collisions via *double hashing* in which a second hash function, different from the main hash function, creates the step size for the key. Then the location of the ticket in the ticket-board is incremented recursively with the step size. Figure 2 shows an example of inserting two pairs into the table.

Retrieval (Lookup): To retrieve values for a set of retrieval queries, each thread is assigned to find the value for one query key. The thread hashes the key to locate the potential entry and first checks its availability bit to see if the table bucket is occupied at all. If the table bucket is not occupied, the key has not been inserted. If the availability bit is unset, the thread compares the info bits that reside beside the availability bit with the info bits from the query key. If they do not match, thread *rehashes* the query key to look for another potential position. Otherwise, it is possible that the key is hashed to the bucket. At last, the bucket in the table is queried and in case it does not match the query key, the key is rehashed with the second hash function. In summary, a retrieval operation in *Stash* introduces three levels of certitude to find a key. First checking the availability bit, second verifying info bits, and third examining the actual bucket content. The number of info bits becomes important in the second level. Choosing a larger ticket size increases the probability to quickly conclude that the bucket does not hold the queried key without having to proceed to the third level. Note that when the ticket size is one (i.e., there are no *info* bits), the second level of retrieval is eliminated.



(a) Stash.



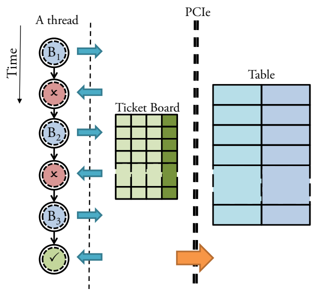
(b) Cuckoo GPU.

Figure 3. Cuckoo GPU uses chaining as a result retrievals might wrongly fail. Stash employs probing, it does not modify hash table during insertions allowing concurrent insertions and retrievals.

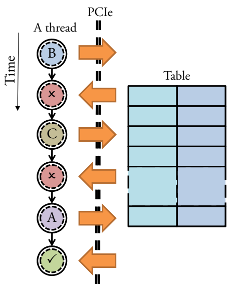
Concurrent Execution of Mixed Operations: Stash allows concurrent insertions and retrievals by employing *probing* as the collision resolution approach. As we mentioned earlier, *chaining* disallows concurrent execution of insertions and retrievals since retrievals can wrongly fail due to temporary removal of key-value pairs from the hash table. On the other hand, with *probing*, during an insert, *ticket-board* entries are probed, first using an initial hashing function and then (repeatedly, if needed) using a rehashing function, until an available ticket (empty table bucket) is found. Since the reservation of the available ticket is performed atomically, and also no available information is extracted neither from the ticket-board nor from the table during the insertion, concurrent retrievals are permitted safely. Figure 3 shows an example of concurrent insertion and retrieval in Stash and Cuckoo GPU.

Efficient Out-of-core Performance: The split design reduces costly accesses to the table entries when the table is very large and is kept inside the host memory. *Stash* keeps the compact *ticket-board*, which is much smaller than the table, inside the GPU global memory. *Stash* tries to resolve the collisions during insertions and mismatches during retrievals using the *ticket-board* before writing the key-value pair into the table or retrieving the pair from the table. During insertions, the table position is reserved inside the ticket-board and the expensive write to the table (inside the host memory) happens only once. Also, during a retrieval, when availability bit is unset, a mismatch between the *info* bits obtained from the key and the ticket *info* bits indicates that the table entry does not contain the requested key-value pair; therefore an unnecessary access to the *table* is avoided. Note that this is a significant improvement compared to GPU Cuckoo hashing [2], which has to directly access the table again each time a collision or a mismatch happens. Figure 4 shows host and global memory transactions for insertion and retrieval in Stash and Cuckoo GPU when the table is outside the GPU global memory inside the host memory.

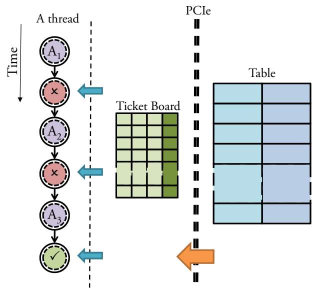
In addition, *Stash* applies atomic operations only on ticket-board positions, unlike existing solutions that perform atomics directly on table entries. *Stash* employs atomic AND to unset (zero) the availability bits in the ticket-board for insertions, regardless of the key or value types in the table. This feature allows having arbitrarily large keys and values



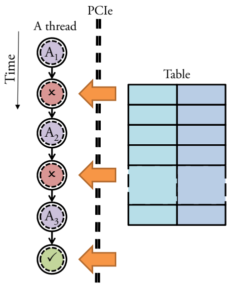
(a) Stash insertion.



(b) Cuckoo GPU insertion.



(c) Stash lookup.



(d) Cuckoo GPU lookup.

Figure 4. Side-by-side demonstration of out-of-core memory transactions by Stash and Cuckoo GPU. Stash reduces costly host memory accesses by using the ticket-board.

eliminating another restriction in existing methods. Next, we present Stash implementation in CUDA.

A. *Efficient Hash and Rehash Functions*

For hashing purposes, we use a generalized function that utilizes a number of high throughput CUDA arithmetic instructions. As lines 1-6 in Figure 5 show, multiple rounds perform reversible transformations on the value initialized with the key. This function is non-linear and provides diffusion in both left and right directions with the shift and the multiplication operations respectively.

For initial hashing of received keys, we employ the function given in lines 7-10 in Figure 5. We first use the general hash function with 4 rounds and a multiplication operand with irregular bits. Then since the general hash function is reversible, i.e. does not reduce the number of possible results, in order to map the outcome of general hash function from 32 bits (`uint`) boundary to table buckets we can use the fast `__umulhi()` function (line 9). Hash functions used in current GPU solutions use *modulo* operator which results in a long instruction sequence on the device. In contrast *Stash* hash functions only rely on high throughput CUDA instructions that translate into fewer assembly operations.

Stash's collision resolution strategy is double hashing thus a second hash function is used to create the step size for the key and move to the next bucket. Every prime number is co-prime with any smaller non-zero integer, therefore, by slightly increasing the size of the table to be a prime

```
1.  template <typename keyT, uint nRounds,
     uint rShift, uint mulOp>
     __device__ uint Hash( const keyT key ){
2.    keyT x = key;
3.    for( uint j = nRounds; j > 0; --j )
4.      x = ( ( x >> rShift ) ^ x ) * mulOp + j;
5.    return (uint)x;
6.  }
7.  template <typename keyT>
     __device__ uint InitHash( keyT key, const uint tableSize ){
8.    uint hashed = Hash<keyT, sizeof(keyT), 8, 0xE9D58A6B>(key);
9.    return __umulhi( hashed, tableSize );
10. }
11. template <typename keyT>
     __device__ uint Rehash( keyT hashed, const keyT key,
     const uint tableSize ) {
12.   uint h_2 = Hash<keyT, sizeof(keyT), 8, 0x6E5B9D8A>(key);
13.   uint dh = hashed + 1 + __umulhi( h_2, tableSize - 1 );
14.   return ( dh >= tableSize) ? ( dh - tableSize ) : dh;
15. }
```

Figure 5. Generalized hash function, initial hashing function, and rehashing function implemented in CUDA.

```
1  hashed := hash key for the first time
2  tryCounter := 0
3  do
4    gotSeat := try reserve an entry with hashed
5    if( !gotSeat )
6        hashed := rehash key
7        if( ( ++tryCounter ) is equal to the table size )
8           the table is full so abort
9  while( !gotSeat )
10 insert ticket info into the ticket-board
11 insert key value pair into the table using hashed
```

Figure 6. *Stash* basic insertion pseudo-algorithm.

number and mapping the result of the second hash function to be a non-zero integer less than the table size, we make sure that hops can traverse all the entries. Lines 11-15 in Figure 5 present the rehashing function. Using the key and a multiplication operand different from initial hashing function's, generalized hash function yields an integer that is mapped to `uint` boundary (line 12). We map this outcome to a non-zero integer less than the table size and add it to the previous bucket location (line 13). Finally, if the addition overflows the location to a position larger than the table size, we cancel the effect (line 14). Note that this strategy allows Load Factors to be as high as 1.

B. Implementing Operations

In this section, we present the detailed implementations of operations supported by *Stash*. The constants used in pieces of code shown are specific to the ticket size of 4 bits. Constants for other ticket sizes can be similarly inferred.

Insertion: Figure 6 shows Stash insertion procedure. First, the thread that is assigned to the key value pair, hashes the key for the first time (line 1) and initializes the counter (line 2). Then it recursively tries to reserve a table entry (lines 3-9). If the entry is already reserved, rehashing is done on the key (line 6). After a successful reservation, the *info* bits get generated and inserted into the ticket-board (line 10). At last, the key-value pair is inserted into the table (line 11).

```
1.  __device__ uint TryBookASeat( uint* ticketBoard,
     uint tbIndex, uint posInInt ) {
2.    uint permit = 1 << posInInt;
3.    uint auth = atomicAnd( ticketBoard+tbIndex, ~permit );
4.    return ( auth & permit ) ? (~0) : 0;
5.  }
```

Figure 7. Table entry reservation function for ticket size 4.

```
1.  __device__ uint bfi( uint source,
     uint destination, uint start, uint length );

2.  __device__ void InsertTicketInfo( uint info,
     uint posInInt, uint* ticketBoard, uint tbIndex ) {
3.    uint prepTicket = bfi( info, ~0, posInInt+1, 3 );
4.    atomicAnd( ticketBoard+tbIndex, prepTicket );
5.  }
```

Figure 8. Ticket *info* insertion function for ticket size 4.

The entry reservation function is shown in the Figure 7. The hashed value's corresponding element index in the ticket-board (`tbIndex`) and the availability bit position inside the ticket-board's 4-byte long element (`posInInt`) are provided for the function from the caller. Inside the function, the `permit` variable holds one set bit at the position of ticket-board's ticket availability bit (line 2). Via an atomic operation, we bitwise AND the ticket-board entry that holds the ticket with the bitwise negate of the `permit`, which has only one zero bit at the position of availability bit (line 3). After the atomic operation, the availability bit will be unset. It is only through the atomic operation's returned value that the thread finds out if it has succeeded in reserving the entry or if the entry had already been reserved by another thread earlier. Using a bitwise AND, thread can verify the availability bit content of the atomic returned value (line 4). Utilizing atomic operation for entry reservation protects a ticket from being reserved for multiple keys. Since reading and modifying the ticket happens atomically, when multiple threads try to reserve a ticket, only one of them will have the return value that indicates a successful booking.

To generate ticket *info* quickly, we use a form of generalized hash function with a different multiplication operand. Figure 8 shows the ticket info insertion function. Line 1 shows the declaration of the wrapper function for utilizing high-throughput PTX bit-field insertion instruction. This function returns the result of inserting `length` bits of `source` starting from `start` bit into a copy of `destination`. We have used this function in our ticket info insertion implementation. The ticket info bits need to be transferred to the ticket-board via an atomic operation because concurrent threads might be accessing the same region. Line 4 performs it via an atomic AND. Line 3 prepares the info bits by placing appropriate number of info LSBs into a variable full of set (1) bits.

After a successful reservation, the consequent operations, including filling up the ticket info bits and insertion of the key-value pair into the table, are all writes. As a result,

```
1  hashed := hash key for the first time
2  info := generate ticket info from key
3  tryCounter := 0
4  do
5    seatFound := try find key using info in hashed location
6    if( !seatFound )
7        hashed := rehash key
8        if( ( ++tryCounter ) is equal to the table size )
9            table is fully traversed so key isn't inserted, return
10 while( !seatFound )
11 if( seatFound indicates the key is not inside the table )
12   key is not inserted
13 else
14   retrieve value from hashed location
```

Figure 9. *Stash* retrieval pseudo-algorithm.

the thread need not wait for the updates to finish; after issuing the memory stores it can exploit Instruction-Level Parallelism (ILP) and keep executing further non-dependent instructions [13].

Retrieval (Lookup): Figure 9 presents the retrieval procedure in Stash. For the key in hand, initial hash function is applied (line 1) and info bits are generated and cropped (line 2) before entering the loop that recursively tries to find the key. Inside the loop, we verify if the key is found in the hashed entry or not (line 5). If not, we rehash the key and try again. When the key is found, its corresponding value is recorded (line 14).

Line 1 in Figure 10 is the forward declaration of the wrapper function that exploits high-throughput PTX bit-field extraction instruction. The function extracts a bit-field of `source` starting at `start` with the length `length` and emplaces it into the LSBs of the returned value. The rest of Figure 10 shows how a given key with its hashed value is found. After extracting the ticket and putting it into a variable (line 6), line 9 provides the first level of certitude by checking if the *availability* bit is set at all. In line 12, the *info* bits of the ticket are extracted to be compared with key's *info* bits for the second level of certitude at line 13. At the last level, the *table* entry key is compared with the key in hand (line 14). As can be seen in Figure 10, only when a thread reaches the last level, a table access (host memory access in case of a *large* table) is performed.

Deletion: Deletion of a pair from the table in Stadium hashing happens via discovering the bucket in the table (retrieve) and then atomically setting the corresponding ticket bits in the ticket-board. Note that unlike regular retrievals, deletion includes more iterations after rejecting a position since there is no guarantee that between multiple keys that hash to the same position, the order of deletions would be the same as the order of insertions.

Support for mixed operation concurrency: To support mixed operation concurrency, the design needs an additional bit named *access* bit, beside the *availability* bit, indicating whether the bucket is being accessed or not. For example, a thread that inserts a key-value pair into the table needs to atomically unset this bit alongside the availability bit, and

```
1.  __device__ uint bfe( uint source, uint start, uint length );

2.  template < typename keyT, typename tableT >
    __device__ uint TryFindTheSeat( keyT key,
     uint hashed, uint info, uint* ticketBoard,
     tableT* table, uint tableSize ) {
5.    uint tbIndex = hashed >> 3;
6.    uint ticketHolder = ticketBoard[ tbIndex ];
7.    uint posInInt = ( hashed & 7 ) << 2;
8.    uint permit = 1 << posInInt;
9.    if( permit & ticketHolder ) {
10.     return KEY_NOT_INSIDE_TABLE;
11.   } else {
12.     uint retrievedInfo = bfe( ticketHolder, posInInt+1, 3 );
13.     if( info != retrievedInfo ) return 0;
14.     return ( table[ hashed ].key == key ) ? (~0) : 0;
15.   }
16. }
```

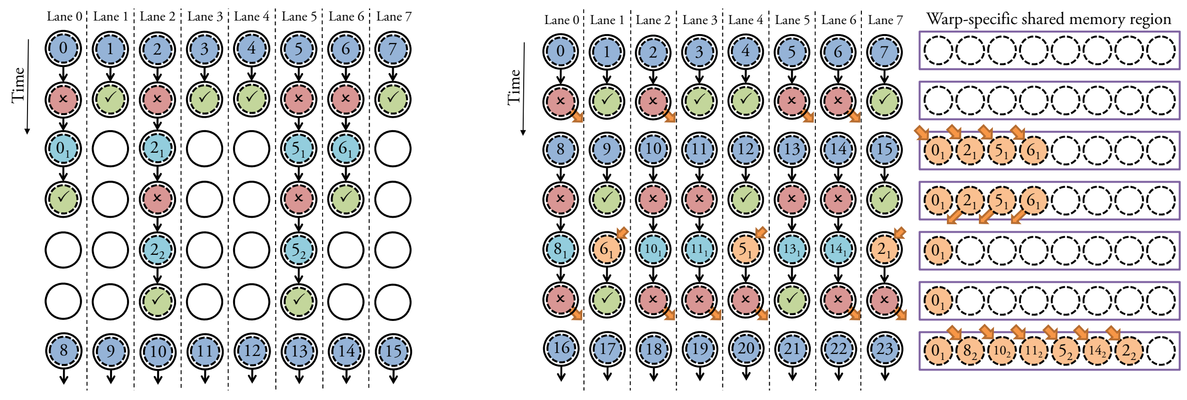
Figure 10. Table entry discovery function for *Stash* retrievals for ticket size 4.

set it when insertion is completed. A proper memory barrier instruction— `__threadfence()` if the table is inside the global memory, `__threadfence_system()` if the table is inside the host memory— will be required in this case after inserting the pair into the table. Stash assumes that there exists a proper implicit or explicit memory barrier between the insertion of a key and its retrieval. Therefore, retrieving or deleting threads ignore the ticket with an unset *access* bit. If there is no memory barrier between the insertion of a key and its retrieval being supplied by the user, retrieving and deleting threads have to wait for the *access* bit of an available ticket to be set again. Also, if deletions and retrievals mix, the retrievals require further iterations after rejecting a bucket.

C. *clStash: Batched Insertions with Collaborative Lanes*

Basic *Stash* explained in the previous section is not aware of GPU's SIMT hardware architecture and actually with slight modifications can be used in other multi-threaded environments as well. In the GPU realm, if a thread is not able to reserve a table entry, it has to rehash its key again hence dragging with itself all other lanes in the same SIMD group (warp) that have successfully reserved entries causing them to stay idle. This dragging phenomenon continues until all the warp lanes find and reserve a table entry, thus lowering the warp execution efficiency. To better utilize GPU resources, we propose *Stash with* collaborative lanes (*clStash*). In *clStash*, each warp uses a portion of the shared memory to collect key-value pairs that could not be inserted into the table at the previous iterations. When there are enough collected pairs to keep all the warp lanes busy, rehashing task is chosen for all of them.

Figure 11 compares *Stash* and *clStash* via an example. Figure 11(a) shows that threads inside a warp are underutilized when only a few threads are not able to find available entries. Figure 11(b) depicts the processing procedure in the collaborative lanes approach. Here shared memory acts as a fast easy-to-access storage for pairs that could not be inserted into the table in the previous iterations. For



(a) Basic Stash. (b) clStash.

Figure 11. Higher warp utilization in Stash with collaborative lanes (clStash) compared to the basic Stash. In the above figure, assumed warp size is 8, and subscript numbers show the number of tries done to insert the pairs into the table.

each warp we select the size of this storage unit equal to the warp size. After every iteration, warp lanes count the number of successfully inserted pairs. If this number is more than the number of pairs collected inside the shared memory–equivalently if the number of pairs that could not be inserted is less than the available warp-specific shared memory capacity, unsuccessful lanes can push their pairs into the storage unit, and then all the warp lanes grab fresh pairs to hash. Otherwise, each successful lane can pop a pair from shared memory, and all threads rehash their pairs. This approach enhances the utilization of warp lanes by postponing processing of some pairs to provide a unique task for all the warp lanes in every iteration.

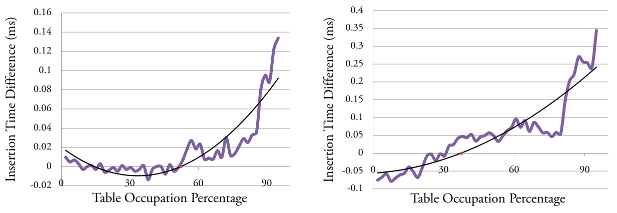
In *clStash*, lanes exploit intra-warp binary prefix sum to identify the address in the shared memory from which they have to extract pairs or to which they have to write pairs. Here the lane success at inserting their pairs in the previous iteration works as the predicate.

1) clStash CUDA Implementation: Figure 12 shows the *clStash* pseudo-algorithm. It starts with initializing a variable that indicates the number of accumulated undone jobs (line 1). The rest of the computation happens in a loop that continues as long as there are more key-value pairs to insert. The thread initializes a job at the beginning of the loop by fetching a fresh key-value pair (line 3). After hashing the key (line 4) and an attempt to reserve an entry for the pair (line 5), successful threads insert the ticket info bits and the pair into the table (line 7 and 8). Since these two steps involve *writing* to the global memory, the warp lanes can hide the latency by executing further non-dependent instructions, avoiding needless idling while the global memory region is being updated. At line 9, we do an intra-warp binary reduction to find how many warp lanes were not successful in finding an empty entry. As long as there are sufficient accumulated undone jobs to be used by those lanes who have successfully finished their job (line 10), warp lanes stay in a

```
1  accumulated := 0
2  while( there exist more key value pairs to hash )
3    fetch a fresh job from global/host memory
4    hashed := hash the key for the first time
5    gotSeat := try reserve an entry with hashed
6    if( gotSeat )
7      insert ticket info into the ticket-board
8      insert key value pair into the table using hashed
9    nUnsuccess := intrawarp binary reduction over (!gotSeat)
10   while( ( nUnsuccess + accumulated ) > warpSize )
11     threadScan := intrawarp binary scan over gotSeat
12     nSuccess := shuffle threadScan from the last warp lane
13     if( gotSeat )
14       address := accumulated - threadScan
15       fetch the job from shared memory using address
16     accumulated := accumulated - nSuccess
17     hashed := rehash the current job
18     gotSeat := try reserve a bucket with hashed
19     if( gotSeat )
20       insert ticket info into the ticket-board
21       insert key value pair into the table using hashed
22     nUnsuccess := intrawarp binary reduction over (!gotSeat)
23   if( !gotSeat )
24     threadScan := intrawarp binary scan over (!gotSeat)
25     address := accumulated - threadScan - 1
26     store the job to shared memory using address
27   accumulated := accumulated + nUnsuccess
```

Figure 12. *clStash* insertion pseudo-algorithm. Note that shared memory declarations are `volatile`.

loop (line 10). Note that this condition is equivalent to *there not being enough storage slots to push unsuccessful jobs.* Inside the loop, successful lanes compute the shared address from which they need to fetch the job via an intra-warp inclusive binary prefix sum (line 11-15). At line 16, all the warp lanes reconverge and start to work on the job they have in their registers. When there are insufficient accumulated undone jobs, the loop (line 10-22) terminates. Now those lanes that were not successful in inserting their pairs, push their jobs into shared memory. Then all warp lanes re-start the outmost loop by fetching fresh jobs from the global memory. Our implementation contains code to take care of what remains in the shared memory after the outmost loop terminates.



(a) Ticket size is 1.

(b) Ticket size is 2.

Figure 13. The measured time difference between *clStash* and *Stash* using 50 evenly distributed samples over insertion of 2^{26} randomly generated pairs. The Load Factor is 0.95, the key and the value each occupy 8 bytes, and the pairs and the table reside in the GPU global memory. The second order polynomial regression for each plot is shown in the same figure.

For a fast computation of intra-warp binary prefix sum and intra-warp binary reduction, we used Harris et al. approach [10]. For aforementioned tasks, [10] efficiently exploits `__popc()` and `__ballot()` CUDA intrinsic functions that translate into one or a few machine instructions.

2) Stash+clStash Hybrid for Batched Insertions: While *clStash* increases warp execution efficiency, it does not come for free. The added overhead of binary prefix sum, counting successful lanes in the warp, checking the overflow condition, and the transactions with the shared memory can make *clStash* slower than the basic Stash insertion when the collision chance is very small. Therefore, we designed a *hybrid* approach in which, when the table is mostly empty, Stash is employed. When the table becomes more occupied, and collisions start to increase, clStash is used.

Hybrid insertion requires a threshold specifying when to switch from the basic approach to collaborative lanes. To determine this threshold, we divided a sample input to many equally-sized chunks and measured the time it takes for every chunk in both approaches to be fully inserted. Then we computed the difference in times of two approaches such that a positive value shows the supremacy of collaborative lanes approach. We interpolated the timing difference plot via a second order polynomial regression and introduced the second in-the-range root as the hybrid threshold. Figure 13 shows this approach for two configurations; thresholds for other configurations were computed similarly.

IV. Experimental Evaluation

We performed our experiments on a system equipped with Nvidia GeForce GTX780 which has 12 GK110 Streaming Multiprocessors and 3 GB of GDDR5 RAM. PCI Express 3.0 operating at 16x speed connects the device RAM and the host DDR3 RAM. Evaluations are performed using CUDA 6.5 on Ubuntu 14.04. Unified Virtual Addressing is in effect and the highest optimization level flag (-O3) is applied during compilation of all programs. In all the experiments, the ticket-board, the input key-value pairs for insertions, the input query keys, and the result array for retrievals reside inside the GPU global memory. All the key-value pairs are generated randomly.

We first compare the performance of Stadium hashing with GPU Cuckoo hashing[2], then examine the benefits provided by our suggested methods, and finally analyze the sensitivity of Stash.

A. Stadium Hashing vs. Cuckoo Hashing

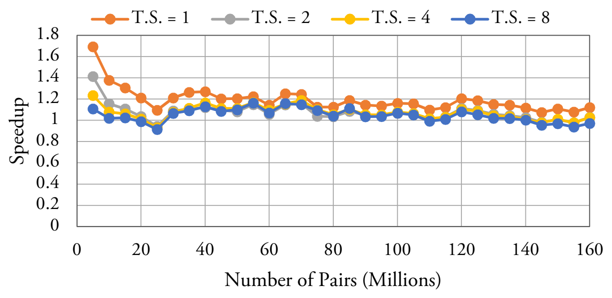
We compare the performance of *Stash* with GPU Cuckoo hashing [2] implemented in CUDA Data Parallel Primitives (CUDPP) Library [11]. Inputs in this section are in the form of interleaved insert and retrieval operations grouped into slices. Queried keys in an interval are selected from the keys inserted in the previous slice. Each slice has approximately 100k insertion or retrieval operations combined. To remove the effect of thread divergence during comparison, we batched insertion and retrievals in groups of 32. In Stadium hashing, each slice is executed by a separate CUDA kernel all inside an stream therefore inserted query keys can be found inside the table during processing the next slice.

As we mentioned earlier, GPU Cuckoo hashing [2] requires the size of the key and the value each to be 4 bytes. Therefore the experiments in this section use 4 byte keys and values. Also, GPU Cuckoo hashing natively does not support concurrency. To enable safe insertion and retrieval for GPU cuckoo hashing inside a slice, we assigned two sequential kernels for each slice: first one performing insertions ignoring retrievals and the second one ignoring insertions retrieving query keys.

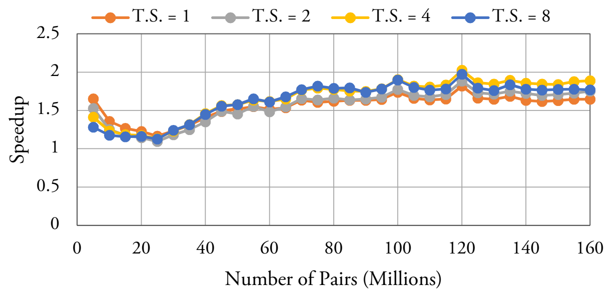
In-core performance: We compare the performance of hybrid Stash with GPU Cuckoo hashing for a range of inputs and presented the speedups using different ticket sizes in Figure 14. When all the query keys exist in the table (Figure 14(a)), our method is 1.04x-1.19x faster than GPU cuckoo hashing on average. When only a portion of the query keys are available in the table (Figure 14(b)), the speedup provided by our method is higher and is between 1.55x-1.65x. The higher speedup range is due to quick rejection of unavailable query keys by the ticket-board.

We further examined the concurrency efficiency of in-core hashing for our method and Cuckoo-GPU. Figure 15 shows the ratio of the aggregation of insertion duration and retrieval duration over combined operations duration. It shows how Stash performance and Cuckoo hashing performance get affected when we combine and interleave Insertions and retrievals. While Cuckoo performance drops significantly by mixing the operations (61.3% on average), Stash performance changes only slightly (concurrency efficiency is 96.8% on average).

Out-of-core performance: Figure 16 presents the speedup of hybrid Stash against GPU Cuckoo hashing when the table is inside host memory. In both Figures 16(a) and 16(b) it is evident that a larger ticket size provides higher speedups due to higher access certainty (lower number of host memory accesses) during retrievals. Also for the inserts, our method incurs only one write-only access to



(a) 100% queries exist.



(b) 50% queries exist.

Figure 14. Stadium hashing speedup compared to GPU Cuckoo hashing. The table is inside the GPU global memory. The number of pairs that have been inserted in insertion phase are asked for during the retrieval. The Load Factor is 0.8.

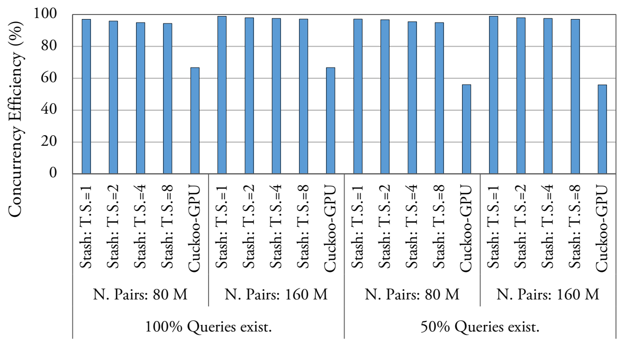
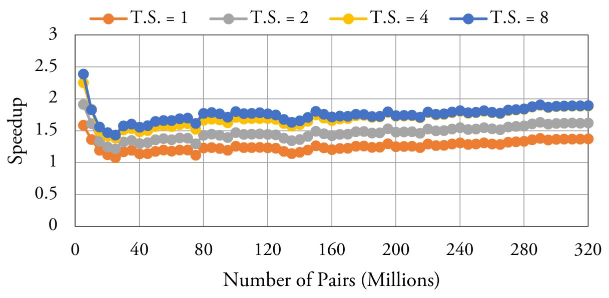


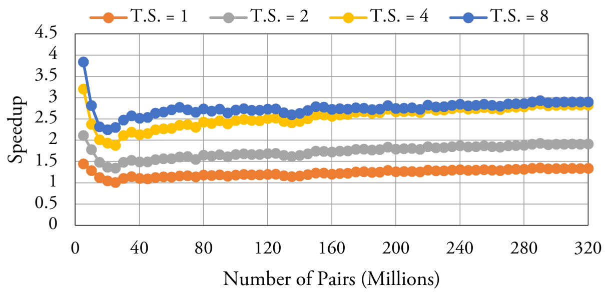
Figure 15. The ratio of the aggregation of insertion duration and retrieval duration over combined operations duration. The table is inside the GPU global memory and the Load Factor is 0.8.

the host memory per insertion request, minimizing costly host memory transactions. When all the query keys are available in the table, our method outperforms GPU Cuckoo hashing by 1.25x−1.75x. When only half of the query keys are available during retrieval (Figure 16(b)), ticket-board helps rejecting unavailable keys without accessing the host memory providing 1.23x−2.47x speedup.

We also compared the exclusive performance of Stadium hashing and Cuckoo-GPU for out-of-core tables in Figure 17. Figure 17(a) shows the results for hybrid *Stash* insertion rates for different ticket sizes. As we can see, regardless of the ticket size, the hybrid *Stash* has approximately twice



(a) 100% queries exist.

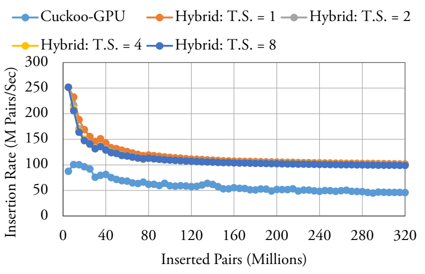


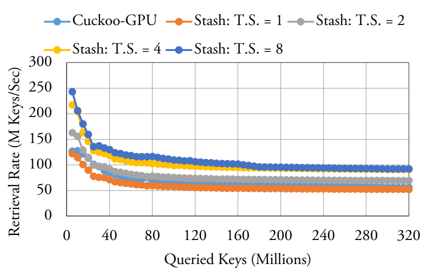
(b) 50% queries exist.

Figure 16. Stadium hashing speedup compared to GPU Cuckoo hashing, having the table inside a pinned host memory region. The number of pairs that have been inserted in insertion phase are asked for during retrieval and deletion. The Load Factor is 0.8.

the performance of Cuckoo-GPU. This is because *Stash* resolves the collisions in the ticket-board and writes to the slow host memory only when it is necessary but Cuckoo-GPU asks for the returned value which adds the overhead of reads from the host side. In Figure 17(b) we show the retrieval rates of *Stash* and Cuckoo-GPU when all the query keys are available inside the table. We observe that as we increase the ticket size and hence ticket info bits, the retrieval rates increase since positions that do not belong to the query key can be rejected in greater numbers and thus avoiding costly table lookups for the actual key. On average, retrievals with ticket sizes 8, 4, and 2 are respectively 1.59, 1.50, and 1.14 times faster than Cuckoo-GPU. Figure 17(c) presents the retrieval rates when only half of the query keys are available inside the table. While *Stash* reduces the number of costly unnecessary PCIe transactions by examining the tickets in the ticket-board, Cuckoo-GPU has to verify all 4 possible entry locations, introducing 4 expensive PCIe reads for unavailable keys. On average, *Stash* retrievals in this scenario with ticket sizes 8, 4, and 2 are respectively 3.99, 3.23 and 1.59 times faster than Cuckoo-GPU.

SIMD execution efficiency: We measured the warp execution efficiency of insertions in Cuckoo-GPU, *Stash*, and *clStash* for various Load Factors and presented them in Table I. In comparison to *Stash* and Cuckoo-GPU [2], the *clStash* strategy greatly improves the warp execution efficiency due to its awareness of underlying SIMD architecture.


(a) Insertion.


(b) Retrieval: all the query keys are available.

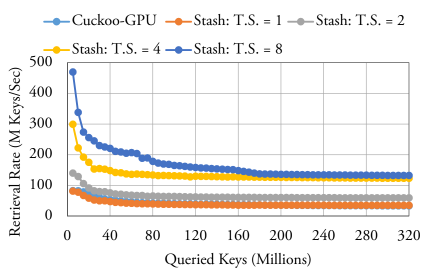
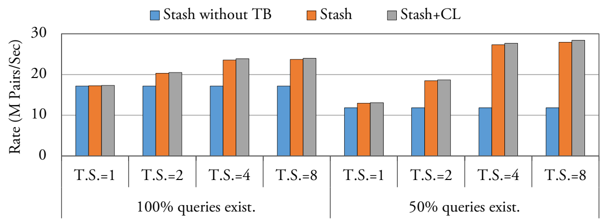

(c) Retrieval: half of the query keys are available.

Figure 17. The out-of-core performance of Stadium hashing in comparison with Cuckoo hashing on GPU [2]. The same number of pairs that have been inserted in insertion phase are asked for retrieval and deletion. The Load Factor is 0.8 in all cases.

	LF = 0.8	LF = 0.85	LF = 0.9	LF = 0.95
Cuckoo-GPU	25.45%	24.5%	23.5%	22.4%
Stash	49.45%	47.0%	44.2%	40.6%
clStash	87.95%	87.45%	86.85%	86.1%

Table I
PROFILED WARP EXECUTION EFFICIENCIES FOR CUCKOO-GPU AND STADIUM HASHING FOR INSERTION OF 2^{25} PAIRS WITH VARYING LOAD FACTORS (LFS). FOR STADIUM HASHING THE TICKET SIZE IS 4.


(a) There are 320M pairs and the table resides in the host memory.

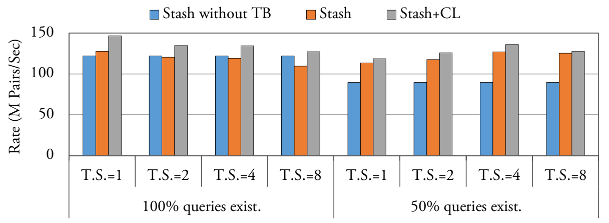

(b) There are 160M pairs and the tables is inside the GPU global memory.

Figure 18. The rate to insert and retrieve key-value pairs when utilizing ticket-board and the collaborative lanes method. The key and the value each occupy 8 bytes. The Load Factor is 0.85.

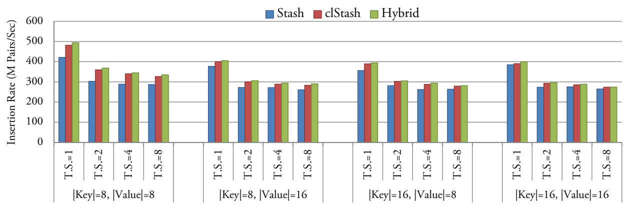
B. Stadium Hashing Performance Breakdown

In this section, we have measured the performance gain provided by employing the ticket-board and the collaborative lanes approach in different scenarios and provided the results in Figure 18. For this purpose we have implemented a version of Stash in which the requests for retrievals are processed by directly comparing the query key with the key in the table, avoiding the ticket-board in retrievals altogether. Figure 18(a) presents the processing rate for Stash and Stash+CL (hybrid) methods and compares them with Stash that avoids ticket-board during retrievals, for when the table is inside the host memory. It is clear that ticket-board especially with larger ticket sizes can help improving the processing rate. This is due to higher rate of unnecessary host memory access avoidance with larger ticket *info* bits about the key. This effect becomes very significant when only a portion of the keys reside inside the table. On the other hand, since the PCIe bandwidth is the bottleneck when the table is in the host side, increasing SIMD efficiency in this case is not very effective. Also when the table resides inside the GPU global memory (Figure 18(b)), enhancing warp execution efficiency of the Stash with Collaborative Lanes method increases the performance.

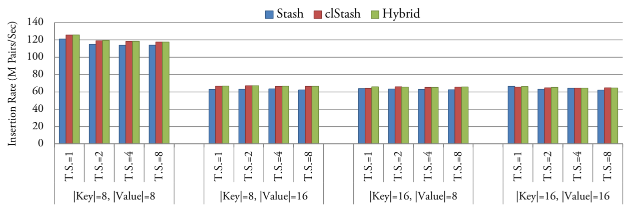
C. Stadium Hashing Sensitivity Analysis

In this section, we analyze the sensitivity of Stadium hashing to the size of keys and values. Figure 19 presents the performance of *Stash* with different ticket sizes over different key and value sizes for in and out of core insertions, retrieval, and deletions. Figure 19(a) plots the insertion rate for Stash, clStash and Hybrid method when the table resides inside the GPU global memory. First, the ticket size one has a higher rate of insertion since it does not involve creation and insertion of ticket info bits. Also, the performance slightly reduces as we increase the key or the value size due to wider memory transactions. Figure 19(b) shows the same scenario having the table inside the host memory. Here the PCIe bandwidth limits the insertion rate in all cases.

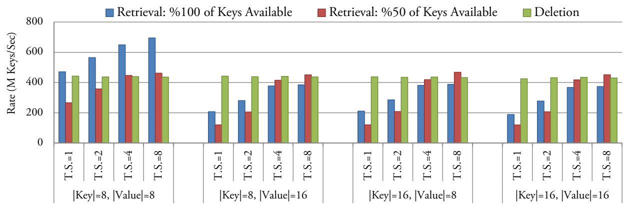
Figure 19(c) shows the retrieval and deletion rates of different ticket sizes facing different key or value sizes. Generally, bigger keys or values require wider memory transactions which bring down the retrieval rate. When the table is in host memory (Figure 19(d)) larger ticket sizes become more necessary to avoid costly host memory transactions, especially with larger key or value sizes.



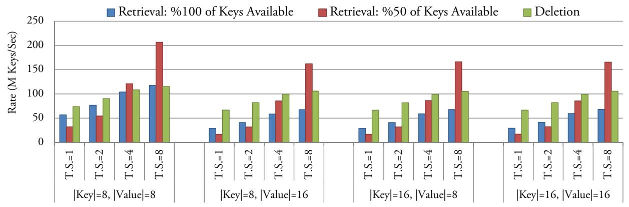
(a) Insertion into the global Mem.



(b) Insertion into the host Mem.



(c) Retrieval and deletion from the global Mem.



(d) Retrieval and deletion from the host Mem.

Figure 19. Sensitivity of Stadium Hashing to different key-value sizes over insertion, retrieval and deletion of 2^{25} pairs or keys. The Load Factor is 0.85 in all cases. Mentioned key or value sizes are in bytes.

V. Related Work

GPU parallel hashing: Parallel hashing on GPUs started to attract attention first for a compact representation of two or three dimensional sparse data: Perfect Spatial Hashing [14] suggests precomputing a prefect hash table for a static set of elements. GPU threads can access the query data through exactly two memory accesses: one to an offset table and one to the hash table. Since the creation of two tables is time-consuming, this solution is helpful only for static data and usually for Graphics applications. Another notable solution is Coherent Parallel Hashing [7] which implements the idea in Robin hood hashing [4]. Similar to Cuckoo Hashing [18], chaining is the collision resolution strategy. Robin hood hashing stores additional bits indicating the age of the key and a key is swapped only if it is younger than another key. Coherent Parallel Hashing yields its best performance only when the keys are neighbors for insertions and accesses are coherent in retrieval. This makes Coherent Parallel Hashing impractical for general purpose applications. To exploit locality, Bordawekar has suggested a multi-level design [3] in which every level utilizes a different hash function and confines the probing region. Nevertheless, this approach imposes restrictions to usability similar to [2]. Other techniques to improve the locality of references are Hopscotch hashing [12] and Cache-Oblivious hashing [19] via bounded probing but in our experiments we observed that primary clustering, especially with larger Load Factors, can outweigh the effect of possibly higher GPU L2 cache hit ratio.

Query processing: Parallel processing of queries on GPUs also involves parallel indexing and retrieval techniques. Bin-Hash Indexing [8] provides a GPU based parallelization method for Query-Driven Visualization. The key space is partitioned into perfect spatial hash tables called bins and only base data of the boundary bins are accessed for candidate checks. Although this technique reduces total global memory accesses, Bin-Hash requires the table data to be pre-encoded on the host side before being transferred to GPU. Diamos et.al. [6] implement several primary relational operators on GPUs using binary search; nonetheless their usage of sorted lists makes incremental insertions very costly. Red Fox [22] proposes a more complete framework consisting of compiler and run-time components for executing relational queries on GPUs. Tables are stored as key and value tuples on padded GPU global memory and manipulated via low level tuple operators (i.e. kernels). Although Red Fox provides a complete infrastructure, it fails to optimize table look-ups, resulting in increased global memory contention.

Load imbalance & branch divergence: The load imbalance problem for GPU hashing can be considered as a subset of a bigger well-known problem for SIMT devices named branch divergence. To remedy the effect of branch divergence, Dymaxion [5] offers *data restructuring* and Zhang et al. suggest *data reordering* [24] but proposed solutions have limited usage to predictable data patterns. *Iteration delaying* [9] is another technique that by means of intra-warp binary reduction, tries to reduce the effect of branch divergence by taking the path most warp lanes will be active in. Unlike Collaborative Lanes technique, *iteration delaying* is unsuccessful in utilizing all the warp lanes in every iteration since no job accumulation strategy is devised.

Parallel prefix sum, employed by Collaborative Lanes, has shown to be a successful approach to balance the load for GPU threads in graph traversal [15]. Furthermore, [17] introduces several techniques to handle load imbalance among which *Local Worklists* method saves a local work queue for each thread in the shared memory; in Collaborative Lanes approach we efficiently accumulate only undone jobs that haven been unsuccessful in previous iterations. Finally, collaborative lanes can be viewed as a hybrid of data-driven computation and topology-driven computation– classified in [16]– that has a partition size equal to the warp size.

VI. Conclusion

In this paper, we presented Stadium hashing (Stash), a parallel hashing method on GPUs that, unlike previous techniques, supports efficient concurrent mixed operations by multiple simultaneously operating threads, effectively enables scaling the hash table to exceed the GPU limited global memory, and does not limit the size of the key nor the size of the value. We then introduced Stash with Collaborative Lanes (clStash) that enhances the GPU warp utilization and further improves the insertion performance. We showed that Stadium hashing can outperform existing GPU parallel hashing methods by up to 2 and 3 times for in-core and out-of-core tables respectively.

Acknowledgment

This work is supported by NSF Grants CCF-0905509, CNS-1157377, CCF-1318103, and CCF-1524852.

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time Parallel Hashing on the GPU. In *ACM SIGGRAPH Asia*, pages 154:1–154:9, 2009.

[2] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, Morgan Kaufmann Publishers Inc., pages 39–53, 2011.

[3] Rajesh Bordawekar. Evaluation of Parallel Hashing Techniques. In *GTC*, 2014.

[4] Pedro Celis, P.-A Larson , and J.Ian Munro. Robin hood hashing. In *Symp. on Foundations of Computer Science*, pages 281-288, 1985.

[5] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems In *SC*, pages 1–11, 2011.

[6] Gregory Diamos, Haicheng Wu, Jin Wang, Ashwin Lele, and Sudhakar Yalamanchili. Relational Algorithms for Multi-bulk-synchronous Processors. In PPoPP, pages 301–302, 2013.

[7] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent Parallel Hashing. *SIGGRAPH Asia*, pages 161:1–161:8, 2011.

[8] Luke J. Gosink, Kesheng Wu and, E. Wes Bethel, John D. Owens, and Kenneth I. Joy. Bin-Hash Indexing: A Parallel Method For Fast Query Processing. Laurence Berkeley National Laboratories, 2008.

[9] Tianyi David Han, and Tarek S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *GPGPU-4*, pages 3:1–3:8, 2011.

[10] Mark Harris, and Michael Garland. Optimizing parallel prefix operations for the Fermi architecture. In *GPU Computing Gems Jade Edition*, Morgan Kaufmann Publishers Inc., pages 29–38, 2011.

[11] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. CUDPP: CUDA data parallel primitives library. http://cudpp.github.io, 2007.

[12] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In *DISC*, pages 350–364, 2008.

[13] Hyesoon Kim, Richard W. Vuduc, Sara S. Baghsorkhi, Jee-Whan Choi, and Wen-mei W. Hwu. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). In *Synthesis Lectures on Computer Architecture*, 2012.

[14] Sylvain Lefebvre, and Hugues Hoppe. Perfect Spatial Hashing. In *SIGGRAPH*, pages 579–588, 2006.

[15] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *PPoPP*, pages 117–128, 2012.

[16] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *IPDPS*, pages 463–474, 2013.

[17] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. In *PPoPP*, pages 147–156, 2013.

[18] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *Journal of Algorithms*, pages 122–144, 2004.

[19] Rasmus Pagh, Zhewei Wei, Ke Yi, and Qin Zhang. Cache-Oblivious Hashing. In *Algorithmica*, pages 864–883, 2014.

[20] T. Sorensen, G. Gopalakrishnan, and Vinod Grover. Towards Shared Memory Consistency Models for GPUs. *ICS*, pages 489–490, 2013.

[21] Nicholas Wilt. The CUDA Handbook: A Comprehensive Guide to GPU Programming. *Pearson Education*, pages 127–128, 2013.

[22] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In CGO, pages 44:44–44:54, 2014.

[23] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. GPU-Accelerated Text Mining. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[24] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *ASPLOS*, pages 369–380, 2011.

[25] Yuan Zu, and Bei Hua. GLZSS: LZSS Lossless Data Compression Can Be Faster. In *GPGPU-7*, pages 46:46–46:53, 2014.