A GPU-Friendly Skiplist Algorithm

Nurit Moscovici, Erez Petrank
Computer Science Dept.
Technion
Haifa, Israel
nurits86@gmail.com, erez@cs.technion.ac.il

Nachshon Cohen
Computer Science Dept.
EPFL
Lausanne, Switzerland
nachshonc@gmail.com

***Abstract*—We propose a design for a fine-grained lock-based skiplist optimized for Graphics Processing Units (GPUs). While GPUs are often used to accelerate streaming parallel computations, it remains a significant challenge to efficiently offload concurrent computations with more complicated data-irregular access and fine-grained synchronization. Natural building blocks for such computations would be concurrent data structures, such as skiplists, which are widely used in general purpose computations. Our design utilizes array-based nodes which are accessed and updated by warp-cooperative functions, thus taking advantage of the fact that GPUs are most efficient when memory accesses are coalesced and execution divergence is minimized. The proposed design has been implemented, and measurements demonstrate improved performance of up to 11.6x over skiplist designs for the GPU existing today.**

***Keywords*-GPU; SIMD; Data Structures; Skip List**

I. Introduction

In recent years, GPUs have become widely available at a low cost. Todays GPUs provide hundreds of computing cores at high energy efficiency, with more cores added in every generation. The introduction of specialized parallel programming platforms such as CUDA [1] and OpenCL [2] over the past decade have opened GPUs for general-purpose programming (GPGPU), without need for a background in computer graphics. Interest in GPGPU has surged in recent years, and GPUs are used today to accelerate applications in a wide variety of fields from deep learning [3] to database operations [4]. However, the design and implementation of efficient general-purpose algorithms remains a significant challenge.

GPUs are very effective for regular-access data-parallel computations on large datasets, often utilizing large vectors or matrices. However, irregular access to memory and control-flow divergence in applications can severely impair performance [5], [6]. These behaviors are often exhibited by pointer-based data structures that support dynamic updates and accesses, which are frequently required in general purpose algorithms. While many such data structures have been developed for use on the CPU, attempts to port them directly to the GPU have shown that further GPU optimizations are necessary [7], [8]. We believe that GPGPU will be able to provide complex services for the CPU in the future, e.g., JIT compilation and garbage collection. To achieve such tasks, we first need to build the basic blocks, and we focus on important data structures, in this case, the skiplist.

Skiplists are popular in concurrent algorithms, as they offer a probabilistic alternative to balanced search trees without costly balancing operations. They have been used as a basis for key-value stores [9], [10] and for other data structures such as priority queues [11]. However, classic skiplist designs provide little locality of data and have highly irregular access patterns, both of which are significant drawbacks on the GPU in terms of performance. Additionally, thread-level synchronization on the GPU is very costly, especially when necessary between any pair of threads in the system.

In this paper we propose GFSL, a GPU-friendly design for a fine-grained lock-based skiplist. GFSL consists of linked lists of array-based nodes, each of which contains several consecutive keys. Threads in a warp access these nodes in a coalesced fashion and cooperate in the execution of each skiplist operation. As such, we reduce the amount of concurrent skiplist operations to gain higher memory coalescence and lower execution divergence, thus playing to the strengths of the GPU.

We compare GFSL to an implementation of a lock-free skiplist algorithm running on the GPU written by Misra and Chaudhuri [7], which was shown to achieve a speedup over the CPU implementation. Results show that our optimizations offer a performance boost for large key ranges. In a range of 10M keys, our implementation offers a speedup of 6.8x-11.6x.

II. Preliminaries

A. GPU And The CUDA Programming Model

This work was designed and implemented in Nvidia's CUDA C++ programming model [1]. CUDA provides SPMD behavior using GPU-side functions called *kernels*. Kernel code is executed in parallel on each of the threads launched by the user. These threads are subdivided into *blocks*, which are distributed amongst the GPU's Streaming

This work was supported by the Israel Science Foundation grant No. 274/14

Multiprocessors (SMs). The SMs are the computational engines of the GPU, and execute the blocks in parallel. When a block terminates, the SM receives and executes a new block until all blocks have been handled.

The SMs further logically subdivide the blocks into units called *warps*, which are the basic unit managed and scheduled by the SM. Threads in a warp share a program counter and proceed through kernel code in lockstep (The SIMT programming model). Warps on an SM are interleaved in order to hide latency. In every cycle the scheduler chooses a warp that is not stalled (e.g., due an in-process memory transaction), and executes its next instruction. On all existing Nvidia GPUs warps consist of 32 threads, though this may be subject to future changes.

B. Considerations For Efficient GPU Programming

While GPUs have the potential to accelerate many kinds of computations, they are not a good fit for every program. GPUs are best suited for computations that can be run on a large number of data elements in parallel. Additionally, the high cost of data transfer must be justified by executing sufficient operations on the GPU for each launch. We present some well known [6] important considerations for efficient programming in the GPU environment.

a) Synchronization: Communication between threads residing in separate blocks is costly, as it can only be performed via the slow global device memory. CUDA supports a variety of atomic operations which can be used for synchronization [12]; however, simultaneous atomic operations by threads in a warp to the same destination are serialized, and will cause the warp to stall until all have completed. Thus synchronizations must be used sparingly and carefully in order to avoid a drop in performance.

Communication between threads within the same warp is achieved more efficiently by utilizing specialized intra-warp operations, supported by CUDA for compute capabilities 3.0 and higher. Two such operations are `_shfl(var, tId)`, which returns the value of a variable held by a thread at the specified channel within the warp, and `_ballot(bool)`, in which each thread offers a boolean value and receives a 32 bit word comprising a corresponding flag bit for each thread in the warp. Such operations must be used with care, as execution divergence causes threads not in the active branch to return default values, possibly with unintended results.

b) Memory Coalescing: A major consideration for improving performance is memory access optimizations [6]: the number of global memory operations in a *kernel* should be minimized and coalesced into the fewest possible transactions. Each half of a warp (*half-warp*) issues access requests separately, and a memory transaction is performed for every cache line covered by the requests. Thus, if all threads in a half-warp access values that can be coalesced into the same cache line then only one memory transaction will occur, while scattered access results in multiple serial transactions. The warp blocks until all transactions are completed.

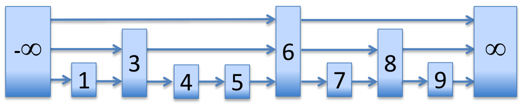


Figure 1. A classic skiplist structure

c) Divergence: If kernel execution causes threads in a warp to *diverge* by executing different branches, all branches will be executed one by one (serially) by the entire warp. Threads that should not be active in the currently executed branch will be temporarily disabled. Thus divergence within a warp may have a negative impact on performance. Additionally, divergence can cause more serious issues in terms of correctness. For example, spin-locks that work correctly in CPU code may cause a deadlock on the GPU when one thread in a warp holds the lock, but the code branch for the spinning threads is performed before the locking thread's branch, causing them to spin forever.

C. Skiplists

Skiplists are widely-used probabalistically balanced search structures that support expected *O(logn)* time for online *Insert*, *Delete* and *Search* operations in ordered collections. While balanced binary search trees offer these results in the worst case, the localized balancing operations required by skiplists make them easier and more efficient to implement in a multithreaded environment [13]. Many concurrent skiplist algorithms exist [14]–[16], though none have yet been designed with GPU-oriented optimizations.

A skiplist consists of layers of sorted linked lists, as in Fig. 1. The bottom level holds all elements in the collection, and every other is a sublist of the level below, containing a random set of keys chosen with some fixed probability p_{key}. Each element receives a random height upon insertion and is linked in every level up to that height. Traversal is performed by searching through each level from the top down, using each lateral step in the higher levels to skip over several keys in the bottom level.

Some skiplist properties make efficient porting to the GPU a challenge. Skiplists have little locality of data, causing slow uncoalesced memory access on the GPU. Skiplist operations also present a high probability for divergence of threads within the same warp: each thread that operates on a different key will have a unique traversal order, potentially causing many branches between the threads. We present a GPU-friendly fine-grained lock-based skiplist design.

III. Algorithm Overview

As discussed above, GPU algorithms are most efficient when performing coalesced memory accesses with low control flow divergence. We tune the classic skiplist structure to

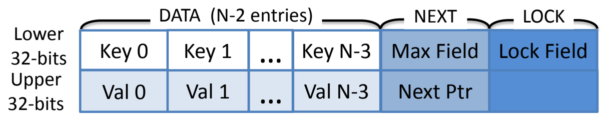


Figure 2. Format of a chunk of size N

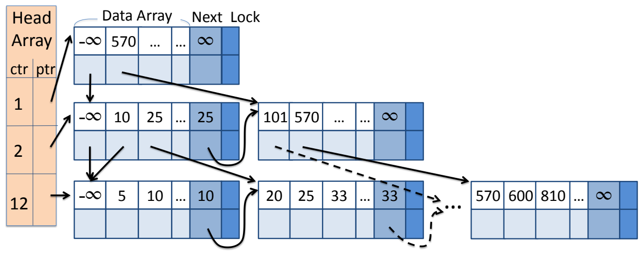


Figure 3. A chunked skiplist

these requirements by using array-based skiplist nodes and allowing threads in a warp to cooperate in the execution of the skiplist operations.

We tackle the problem of scattered memory accesses by packing consecutive key-value pairs residing in the same level into large cache-aligned skiplist nodes called *chunks*, shown in Fig. 2. Chunks contain a *data array*, a sorted array of key-value pairs, along with a LOCK entry and a NEXT entry consisting of a pointer to the next chunk and a max field holding the maximum key in the current chunk. hunks are designed to be read efficiently in the fewest possible memory transactions.

GFSL consists of several levels of chunked linked lists, each containing a subset of the keys in the level below, as seen in Fig. 3. Each chunk's data array is sorted in rising order, with empty entries denoted by a special ∞ value and grouped at the end of the array. In the upper levels the value field of each entry in the data array points to a chunk in the level below, and in the bottom level this field will hold the data element associated with the corresponding key. A key-value pair in level $i+1$ generally points to a chunk containing the same key in level i, though it may temporarily point to a chunk containing smaller values during *Inserts* and *Deletes*. The first chunk in each level contains a $-\infty$ key in the first entry with a pointer to the first chunk in the level below, and is accessed via a pointer from the *Head Array*. The last chunk in every level contains an ∞ value in both its next-pointer and max fields. ∞ and $-\infty$ are distinct from keys in the structure.

Threads are divided into groups called *teams*, which cooperate to perform the skiplist operations. Teams can be defined by the user to be either the size of a warp or smaller. The number of entries in a chunk is equal to the number of threads in a team, so that the entire chunk is read in a single kernel instruction (executed in lockstep by the team). Each thread in a team simultaneously reads data from the chunk index corresponding to its place within the team (tId). For a team of size N the first *(N-2)* threads, called DATA threads, access the data array, while the last two access the NEXT and LOCK values respectively. Each thread performs computations on the value it read then cooperates with the rest of its team to decide on the next step in the execution via intra-warp operations.

Structure traversal is similar in spirit to traversal over a regular skiplist. A team searching for a key k reads the first chunk in the highest level. Each DATA thread compares k to the key read from its entry, while the NEXT thread compares k to the maximum field. The threads share their results and decide simultaneously how to continue the traversal: either a lateral step via the next pointer, or a step down to the next level via a pointer in some DATA field. The team continues laterally if the searched key is greater than the maximum and steps down otherwise via the data-entry containing the largest key smaller or equal to k. If all keys in the chunk are greater than k then the team must backtrack to the previous chunk in the level and step down from there.

Insert and *Delete* operations are likewise performed by an entire team in tandem while ensuring the chunks remain both internally and externally sorted. If an insertion occurs when there is no free space in the data array a *split* operation is performed: A new chunk is allocated and added to the structure after the *overflowed* chunk. The data array is divided equally between both chunks, whilst remaining sorted. Conversely, if a deletion causes a lower bound on the number of key-value pairs to be crossed then a *merge* operation is performed: the chunk is marked as a *zombie* and its values are moved to the next chunk in the level. If the next chunk is too full this operation may cause it to be split. Pointers are redirected after both split and merge operations in order to ensure the upper level pointers remain accurate and to physically remove a *zombie* from the structure. All changes to the contents of the skiplist are performed under the protection of the chunks' locks, so at most one team can change the contents of a chunk at any time.

GFSL contains fewer nodes and levels than the classic skiplist. A single node in GFSL contains several keys, and so replaces several separate nodes in the classic version. Thus more keys can be inserted into a level before it becomes necessary to add a pointer in the level above. The teams process more data for every memory transaction than a single thread does in the original algorithm, enabling faster traversals over the structure, while also causing less divergence within a warp.

Unlike the classic skiplist algorithm, GFSL does not predetermine a level for every key inserted. Instead, a key can be raised to level $i+1$ only as a result of a split, i.e. when a new chunk is added to level i. Raising the key as a result of insertion of new chunks and not single keys causes the factor between levels to be tied to the number of entries in a chunk, aiding in shorter traversals. In an ideal structure at most one key from each chunk in level i would appear in level $i+1$. In this paper we differentiate between p_{key}, the probability a key in level i will appear in level $i+1$, and p_{chunk}, the probability a key from a chunk in i will be raised to $i+1$

IV. Algorithm Details

A. Structure Details

During the initialization stage we create the structure and allocate an array of chunks in the device memory for a memory pool. The structure initially consists of a single unlocked chunk in each level, containing the $-\infty$ key and a pointer to the chunk in the level below. The head array is initialized to point to these chunks. Each head array pointer is associated with a counter of the number of utilized chunks in the level, initially 0. The counters are used to keep track of the highest level currently used in the structure, and thus to avoid traversal of empty levels.

Allocations from the memory pool are performed by incrementing a global counter and using the resulting index as a pointer. All chunks are allocated locked with ∞ values in all key-data pairs, as well as in the max field. The ∞ max field signifies that this is the final chunk in the level.

Removal of chunks from the structure occurs only during a merge operation. The deleting team marks a chunk as a *zombie* using a special value in the lock field. The *zombie* will eventually be physically removed. While *zombies* are no longer considered to be in the structure they may still be reachable until all pointers to them are redirected. Identifying when a *zombie* is disconnected and can be reclaimed is difficult, as it may be pointed to by multiple chunks.

Memory reclamation is a significant challenge, even on the CPU [17]–[22], often requiring the use of complicated code or locks, which are performance drags on the GPU. A possible reclamation scheme would be to compact the structure between kernel launches; this is also challenging and is left for future work. The need for reclamation in GFSL is reduced significantly compared to Misra and Chaudhuri [7] by the fact that chunk entries from which keys have been removed can be reused as long as the chunk is not a *zombie*.

A chunk is said to *enclose* a key k if it is the first non-*zombie* chunk in the level with a max field greater or equal to k. If k exists in level i it will be found in its *enclosing* chunk in that level. Additionally, k can only be inserted into its *enclosing* chunk.

In this paper we consider chunks with N=32 entries. Each entry is 8B, divided equally between key and value. The small size of keys and values in the structure is necessary as the GPU has a small memory capacity, and memory transfer between the host and device is very slow. Additionally, larger values would require either more transactions or fewer key-value pairs read per transaction. A 32-bit value field may be used to indicate the address of a larger object in the main memory as in Zhang et al. [23].

Table I
Notations

K	Key type. **unsigned int**.
V	Value type. **unsigned int**.
KV	Key-value pair in chunk. **unsigned long**.
CHK*	Pointer to a chunk.
	Special Values
tId	(Thread Idx)%team; //thread's index within its team
DSIZE	$N-2$ - size of data array
NONE	A value distinguishable from any tId

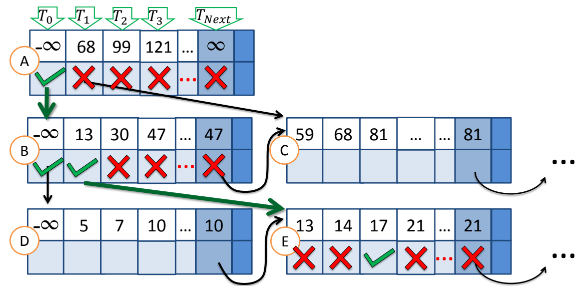


Figure 4. Example of a team performing *Contains*(17).

B. Data Structure Operations

In this section we present the algorithm for the *Insert*, *Delete*, and *Contains* operations in detail. Table IV-B defines some notations. Note that in this section we use CHK* to indicate a pointer to a chunk in global memory in order to simplify the pseudocode. In actuality, chunks are accessed using 32-bit indexes to the memory pool. For chunks of size 128B this index size can cover addresses in 512GB of memory. This is sufficient in the foreseeable future, as modern GPUs have only a few GB of device memory.

1) Contains:

a) General Description: As *Contains* are typically the most common operation called in programs using skiplists, it is vital that the traversal be as fast as possible. A *Contains* operation that must wait for a lock to be released may result in high contention, especially in the massively multithreaded environment of the GPU. Thus, the *Contains* operation is lock-free: it never acquires a lock or waits for a lock acquired by another operation.

A team performing a *Contains* operation searches for a key k, starting from the first chunk in the highest level. The team searches each of the upper levels in turn for the largest key in that level that is smaller or equal to k. Once this key is found the team reads its associated pointer, which is used to step down to the next level. When the bottom level is reached the team begins a lateral search for a chunk containing k itself. A key is considered to be in the structure if it exists in a non-*zombie* chunk in the bottom level.

zombies encountered during traversal are ignored by taking lateral steps until a non-*zombie* is found. The LOCK thread contributes only in recognition of *zombie* chunks, in all other steps decisions are made based solely on the values read by the DATA and NEXT threads.

Consider Fig. 4, in which a team searches for key 17, as a quick example of a *Contains* operation. The team begins by reading chunk A, the first chunk in the top level. Each DATA thread checks whether its value is a candidate for a down step (less than or equal to 17). The NEXT thread checks whether the maximum value in the chunk is smaller than 17, meaning a lateral step should be taken. The team uses `_ballot`, which gives each thread a bitmap mirroring the results of this computation. The threads see that T_0 is the highest thread that returned *true*, and retrieve the pointer from T_0. The entire team then steps down to chunk B, and repeats the computation, finally stepping down into chunk E. In chunk E each DATA thread checks whether its key is equal to 17, while the NEXT thread continues to check whether a lateral step should be taken. The `_ballot` operation shows that T_2 sees key 17, and the team concludes the operation with a *true* indication.

b) Implementation Details: Algorithm 1 shows the *Contains* operation, which calls two main functions. The `searchDown` function, described in Algorithm 2, handles traversal of the upper levels. It begins with calls to the `getHeight` and `firstChunkAtLevel` functions to retrieve the height and a pointer to the first chunk. Both functions are *cooperative*: they utilize intra-warp operations to share data local to each thread. Each thread reads a separate space in the head array to see whether the level corresponding to its *tId* is in use. The team then uses `_ballot` and `_shfl` operations to discover the highest nonempty level and retrieve its pointer.

Algorithm 1. Contains

```
bool contains(K k){
    CHK* pCurr = searchDown(k)
    return searchLateral(k, pCurr)
}
```

In each iteration the team reads a chunk from memory then uses the cooperative function `getTidForNextStep` described below to decide what the next step should be. There are three possibilities for the next step: a lateral step in the same level, a step down to the lower level, or a backtrack through the previous chunk in the same level.

Down steps are demonstrated in the example above. Lateral steps occur when *k* is greater than the maximum key in the chunk. In both cases, `getPtrFromTid`, implemented using `_shfl`, is called to retrieve the pointer in the key-value pair held by the thread with the `tId` chosen as the next step. A backtrack occurs when a lateral step reaches a chunk in which all keys are greater than *k*. In this case the team must step down using the maximum key in the previous chunk. As an example, a team searching for key 50 in Fig. 4 will take a lateral step from chunk B to chunk C. The team will discover that all keys in C are greater than 50, and must step down through key 46 in chunk B. This sequence of operations is similar to the classic skiplist traversal algorithm. To enable this step the team keeps track of the entries read from the previous chunk in the traversal when taking lateral steps (Line 15).

The helper functions called by the `searchDown` algorithm are all cooperative. We consider `getTidForNextStep` as an example of such a function. Other cooperative functions described in this paper are implemented in a similar fashion. Note that `_shfl` and `_ballot` operations are performed by the entire warp. Thus care must be taken to only evaluate values read by the current team when using teams smaller than warp size. In this work only one team is run per warp, regardless the team size[1].

Algorithm 2. searchDown

```
1  uint searchDown(K k){
2  search:
3      KV prevKv = null
4      int height = getHeight()
5      CHK* pCurr = firstChunkAtLevel(height)
6
7      while(height>0) {
8          KV currKv = pCurr−>read(tId)
9          if (isZombie(currKv)) {
10             pCurr = getPtrFromTid(NEXT, currKv)
11             continue
12         }
13         int stepTid = getTidForNextStep(k, currKv)
14         if (stepTid == NEXT) { //lateral step
15             prevKv = currKv
16             pCurr = getPtrFromTid(NEXT, currKv)
17         }
18         else if (stepTid != NONE) { //down step
19             height −−
20             prevKv = null
21             pCurr = getPtrFromTid(stepTid, currKv)
22         }
23         else { //backtrack
24             if(prevKv == null) goto search
25             height −−
26             pCurr = backTrack(prevKv, k)
27         }
28     }
29     return pCurr
30 }
31
32 CHK* backTrack(KV& prevKv, K k){
33     int stepTid = getTidOfDownStep(k, prevKv)
34     CHK* pNextStep = getPtrFromTid(stepTid, prevKv)
35     prevKv = null
36     return pNextStep
37 }
```

In `getTidForNextStep`, shown in Algorithm 3, we see an example of the _ballot operation. Each thread simultaneously calculates a boolean value dependent on its `tId`, *k*, and the key it read from the chunk. The threads then call `_ballot` simultaneously to receive the results of this calculation for each thread. The NEXT thread passes a *true*

[1]We also implemented support for two teams in the same warp performing two different operations in parallel. However, the complexity of the code needed in order to ensure teams within a warp could not deadlock each other caused a degradation in performance.

value to `_ballot` only if k is greater than the max field, and the DATA threads pass a *true* value only if the key they read is less than or equal to k. The LOCK thread always passes a *false* value. Any EMPTY (∞) key value read by a thread will result in a *false* value being evaluated. Thus, the next step required by the algorithm can be decided by taking the highest `tId` that evaluated a *true* flag. This `tId` is determined by subtracting leading zeros (clz) from the ballot return size (32 bits). Precedence is effectively given to threads with higher `tId`s, a fact that is taken into account during *Inserts* and *Deletes* to safeguard against traversals considering bad chunk values. If all threads return *false* then a special NONE value will be returned, signifying a backtrack.

Algorithm 3. getTidForNextStep

```
void getTidForNextStep(K k, KV currKv){
    bool elem = (tId < DSIZE) && (currKv.key <= k)
    bool next = (tId == NEXT) && (currKv.key < k)

    uint bal = (__ballot(next || elem))
    if (bal == 0) return NONE
    return 32 − clz(bal) − 1
}
```

Searching along the bottom level is performed by the `searchLateral` function presented in Algorithm 4. The traversal is very similar to the lateral step in `searchDown`, the main difference being that DATA threads evaluate whether the key they read is equal to k. The team calls `getTidWithKey` to determine the next step, and continues to take lateral steps as long as the NEXT `tId` is returned or the current chunk is a *zombie*. Traversal ends when a value other than NEXT is returned, indicating that the *enclosing* chunk has been reached. The threads finally determine whether the value returned was NONE, indicating that k was not found, or the `tId` of some DATA thread, indicating that k was seen by that thread.

Algorithm 4. searchLateral

```
bool searchLateral(K k, CHK* pCurr){
    do {
        KV currKv = pCurr−>read(tId)
        int foundTid = getTidWithKey(k, currKv)

        if (foundTid == NEXT || isZombie(currKv)) {
            foundTid = NEXT
            pCurr = getPtrFromTid(NEXT)
        }

    } while(foundTid == NEXT)
    return foundTid != NONE
}
```

c) Lock-Freedom: There exists a rare state in which `searchDown` is delayed by a concurrent *Delete* operation and must be restarted, making *Contains* lock-free. We use Fig. 4 to illustrate this edge case. A team searching for key 70 steps from chunk A to chunk C, then stalls. A concurrent team deletes keys 59 and 68 from the structure. When the first team wakes, it sees a chunk containing only keys greater than 70, and so decides to backtrack. As the previous chunk in the new level is unknown, the team does not have enough data to perform the backtrack. The previous chunk in the layer above might also not hold enough information to continue, and so the traversal is restarted. These rare restarts do not limit system progress (they are caused by progress in *Delete* operations), and have a minor effect on measurements (they occur in less than 0.01% of *Contains*).

2) Insert:

a) General Description: The *Insert* function receives $<k,v>$, the key-value pair to be inserted, and searches the structure for k. The insertion is executed only if k is not already in the structure. If insertion causes a chunk overflow a split operation will occur and a new chunk will be added to the structure, containing the top half of the values from the chunk that was split.

The *enclosing* chunk in the bottom level is locked once it is reached and found not to contain k. It remains locked until the *Insert* operation is completed, including all insertions to higher levels. This ensures there are no concurrent *Insert* or *Delete* operations on the same key. In all upper levels the *enclosing* chunk is locked before inserting the key, then immediately unlocked to minimize contention. A key is raised to level $i+1$ only as a result of a split in level i. The decision whether to raise a key after a split is randomly generated (on-device) according to p_{chunk}.

For example, a team executing *Insert*(18) on the structure in Fig. IV-B1a traverses chunks A, B and E, then locks chunk E. The team then inserts key 18 into E. If chunk E is full this causes E to be split. The team will then lock chunk B, perform an insertion, then unlock chunk B. If chunk B is full the team will perform a similar insertion into chunk A. Otherwise the team unlocks chunk E and returns *true*.

b) Implementation Details: The *Insert* function, presented in Algorithm 5, begins by searching for k using the `searchSlow` function, and returns *false* if k already exists in the structure. `searchSlow` performs the same traversal as *Contains*, with two main differences: firstly, `searchSlow` returns the traversal path. The path is made up of the chunks through which down-steps were taken during the traversal, and the *enclosing* chunk in the bottom level. These serve as a starting point for discovering the correct place for insertion in each level. In the example in Section IV-B2a the path would consist of chunks A, B and E. Secondly, when a *zombie* is discovered after a lateral step the team attempts to redirect the previous chunk's pointer to remove the *zombie* from lateral traversals. The redirection is performed lazily by calling try-lock on the previous chunk. If the lock fails the team continues without updating. Update of down-pointers is discussed below.

One would expect a path to be an array of pointers to nodes in each level. However, local arrays are costly in CUDA in terms of resources. Thus, the path is contained in an “artificial array” consisting of a single variable (*path*) per thread. The thread with `tId`=*i* holds the chunk in level *i* in the path. The “array” is accessed using `_shfl` operations. This limits the maximum height of the skiplist to the team size. However, this limit was deemed sufficient, even for teams that are smaller than warp size. For example, chunks of size 16 hold an average of 10 keys. Thus a structure with a maximum height of 16 can be expected to support 10^{16} keys without compromising the skiplist structure. Likewise, chunks of size 32, as shown in the evaluation, allow for around 20^{32} keys. Both are far beyond the global memory capabilities both in current GPUs and those in the foreseeable future.

Algorithm 5. Insert

```
1  bool insert(K k, V v){
2      <bool found, CHK* path> = searchSlow(k)
3      if (found) return false
4
5      bool raiseKey = false
6      CHK* pBottom = getPathFromTid(0)
7      if (!insertToLevel(0, pBottom, k, v, raiseKey)) {
8          unlockChunk(pBottom)
9          return false
10     }
11     v = pBottom
12
13     int level = 1
14     while((raiseKey) && (level < MAX_LEVEL)) {
15         CHK* pEnclose = getPathFromTid(level)
16         insertToLevel(level, pEnclose, k, v, raiseKey)
17         v = pEnclose
18         unlockChunk(pEnclose)
19         level++
20     }
21     unlockChunk(pBottom)
22     return true
23 }
24
25 bool insertToLevel(int level, CHK* pEnc,
26                               K k, V v, bool& raiseKey){
27     pEnc = findAndLockEnclosing(pEnc, k)
28     KV encKv = pEnc−>read(tId)
29     if (chunkContains(encKv, k)) return false
30     raiseKey = false
31     if (numKeysInChunk(encKv) < DSIZE) {
32         executeInsert(pEnc, encKv, k, v)
33         if ((level > 0) && (isLevelEmpty(level)))
34             incrementNumChunksAtLevel(level)
35     }
36     else {
37         <pEnc, k> = splitInsert(pEnc, encKv, k, v, level)
38         incrementNumChunksAtLevel(level)
39         raiseKey = isKeyRaised()
40     }
41     return true
42 }
43
44 void executeInsert(CHK* pEnc, KV encKv, K k, V v){
45     KV insertKv = getChunkValFromLeftNeighbor(encKv)
46     uint insertIdx = getInsertionIdx(insertKv, k)
47     if (tId == insertIdx) insertKv = pair(k,v)
48     for(int i = DSIZE−1; i >= insertIdx; i −−){
49         if ((insertKv.key != EMPTY) && (tId == i))
50             pEnc−>AtomicWrite(tId, insertKv)
51     }
52 }
```

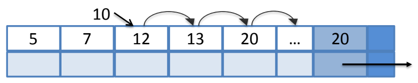


Figure 5. Inserting key 10 into a chunk without a split. All keys higher than 10 are moved one entry to the right.

If *k* was not found, `insertToLevel` (Algorithm 5, Line 25) is called to perform the insertion. `insertToLevel` locks the *enclosing* chunk and inserts *k*, performing a split if necessary, then returns the locked *enclosing* chunk and an indication whether a key should be raised to the next level (Lines 7, 16, and 39). `insertToLevel` will return *false* if *k* was concurrently added by another team before the lock was caught. In lines 11-20 insertion into higher levels is handled by further calls to `insertToLevel`. The value field inserted into level $i+1$ is a pointer to the new chunk in level *i* (Lines 11 and 17).

`findAndLockEnclosing` (Line 27) is a spin-lock that performs a lateral search in order to ensure that the chunk being locked *encloses k*. If the current chunk is a *zombie* or does not *enclose k* the team will read the next chunk. Otherwise the function checks whether the chunk is unlocked before the LOCK thread attempts to lock it using CAS. The team checks whether the lock succeeded, and if so rereads the locked chunk. If the chunk no longer *encloses k* the lock will be released and the team will continue to the next chunk.

`insertToLevel` calculates the number of empty entries in the data array (Line 31). If there are empty entries, `executeInsert` is called to physically insert $<k,v>$, otherwise `splitInsert` is called to split the current chunk and perform the insertion. A level's chunk counter is incremented every time a split occurs or a level is inserted into for the first time.

`executeInsert` (Algorithm 5, Line 44) inserts $<k,v>$ while ensuring the chunk remains sorted. In Line 45 each thread takes the key-value pair from the previous thread in the team using a cooperative function. Then, in Line 46 the insertion index for $<k,v>$ in the sorted data array is determined in another cooperative function. In Lines 48-51 every thread with a `tId` higher than the insertion index writes its neighbor's value into its own place in the data array, thus shifting all entries greater than the new key to the right as shown in Fig. 5. In the same lines, the thread with the `tId` equal to the insertion index inserts $<k,v>$ into the data array.

The insertion is performed serially, from the last DATA index down to the insertion index. In this way we ensure that we do not temporarily cause a key to be overwritten, which may cause a concurrent search to miss an existing key. All search functions polling a chunk for containment of a certain key give precedence to higher threads, and so a

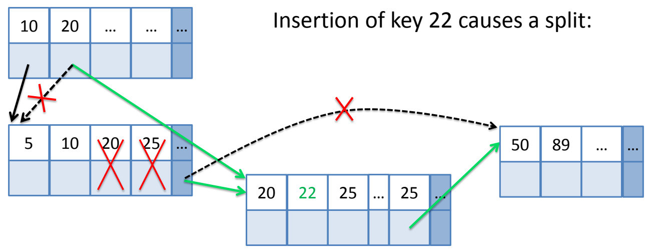


Figure 6. Splitting a chunk. Keys 20 and 25 are moved to the new chunk. The original chunk's next pointer and the down-pointer associated with key 20 in level $i+1$ are redirected to the new chunk.

key temporarily appearing twice in a chunk does not cause search errors. The max field is never changed by such an insertion, from the definition of an *enclosing* chunk.

If the chunk is already full, the team calls `splitInsert` (Algorithm 6) to perform a split as shown in Fig. 6. The `preSplit` function (Algorithm 6 Line 15) locks the next chunk, removing *zombies* if they are encountered (Line 16), then allocates a new chunk which is initialized to point to the next chunk.

Algorithm 6. splitInsert

```
1  <CHK∗, K> splitInsert(CHK∗ pSplit, K k, V v, int level){
2      CHK∗ pNew = preSplit(pSplit)
3      Kv splitKv = splitCopy(pSplit, pNew)
4      CHK∗ pInsert = insertNewData(k, v, pNew, pSplit, splitKv)
5
6      if (pSplit == pInsert)
7          unlockChunk(pNew)
8      else
9          unlockChunk(pSplit)
10     k = keyForNextLevel(k, pInsert, pNew, pSplit, level)
11     updateDownPtrs(level, splitKv, pNew)
12     return <pInsert, k>
13 }
14
15 CHK∗ preSplit(CHK∗ pSplit){
16     CHK∗ pNext = lockNextChunk(pSplit)
17     CHK∗ pNew = alloc()
18     updateNextField(pNew, pNext)
19     return pNew
20 }
21
22 KV splitCopy(Chk∗ pSplit, CHK∗ pNew){
23     KV splitKv = pSplit−>read(tId)
24     K thresh = getKeyFromTid(splitKv.key, DSIZE/2−1)
25
26     if (splitKv.key > thresh)
27         copyToNewChunk(pNew, splitKv)
28     if (tId == NEXT)
29         updateNextField(pSplit, pNew)
30     setMovedValsEmpty(splitKv)
31     return splitKv
32 }
```

`splitCopy` (Algorithm 6 Line 22) is then called to copy the top DSIZE/2 values to the new chunk (Lines 24-27). Once the copy is completed the new chunk can be connected to the structure by redirecting the next pointer of the original chunk and setting its max value to the highest remaining key. Both of these changes are performed with a single atomic write by the NEXT thread (Line 29). The team can then atomically write an empty value to each of the moved values in the old chunk (Line 30). Again we rely on the fact that traversals give precedence to higher `tIds` to argue that a concurrent traversal will not be adversely affected. The updated max field ensures the NEXT thread's value is considered before keys that have not yet been emptied.

The split continues at Line 4 where $<k,v>$ is inserted into the either the old or the new chunk, depending on k's place the sorted array of values. If $<k,v>$ is inserted into the new chunk during a split in the bottom level the original chunk will be unlocked and the new chunk will remain locked until the end of the *Insert* operation, thus ensuring that the *enclosing* chunk in the bottom level remains locked.

The split function determines which key will be raised should the team decide to insert into the next level. As raising a key indicates that a new chunk was created it would make sense to raise the minimum key in the new chunk (*minK*). However, if $k > minK$ then we cannot raise *minK* without performing a new traversal to discover the path to it. Thus, in Line 10, the key raised from level 0 is chosen to be the maximum between k and *minK*. In upper levels the key raised must be the key that caused the split, as the lock on the bottom level protects only keys in that chunk.

Finally, the team updates the down-pointers in level $i+1$ to reflect the changes in level i (Line 11), by searching level $i+1$ for the range of moved keys, then locking affected chunks and atomically updating relevant down-pointers. In Fig. 6, key 20 was moved in the split of level i, causing its down pointer in level $i+1$ to be updated to point to the new chunk. The pointers that have not yet been updated point to legal chunks in terms of traversal, as the *enclosing* chunk can be reached from them using lateral traversal.

3) Delete: The *Delete* operation is similar in spirit to the *Insert*. It begins by searching for the key to be deleted, k, and creating the traversal path in the same way as *Insert* does. If k exists in the structure, the bottom level chunk that *encloses* k is locked. After determining that k is still in the structure, the team searches all occupied levels from the top down, removing k in every level in which it is found. The chunk in the bottom level remains locked until k has been physically removed from all levels, concluding the *Delete* operation. As in the case of *Insert*, this ensures that no other team can concurrently perform updating operations on k.

Removing k from a level is divided into three cases: (1) k can be removed without performing a merge (2) a merge is required (3) k is situated in the final chunk in the level. A merge is necessary if removing k will cause the number of nonempty entries in the data array to cross a predetermined threshold (DSIZE/3 in our implementation).

If no merge is required, the key will be removed in a manner similar to `executeInsert`, though in the opposite direction as illustrated in Fig. 7. Each thread reads the key-value pair of its neighbor to the right with a_shfl operation.

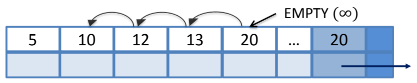


Figure 7. Deleting key 10 from a chunk. All keys greater than 10 are moved one entry to the left.

DATA threads with `tIds` equal to or higher than k's index atomically write their neighbors value into their own index, overwriting the removed key. As in the case of insertion the order of operations matters: the writes must occur from k's index up to the highest DATA `tId` so as not to cause keys to temporarily disappear from the chunk, which could harm concurrent traversals.

There are two cases that must be handled when deleting k that have no equivalent in executeInsert: Firstly, if k was the last element in the chunk the NEXT thread must update the max field. This must occur before the deletion of the key so that concurrent searches do not see a max value that does not exist in the chunk. Secondly, the highest `tId` to see a non-EMPTY value in its entry must make it EMPTY.

If a merge operation is deemed necessary the team locks the next non-*zombie* chunk in the level, redirecting the next pointer to unlink *zombies* if they are found. If the next chunk is too full to receive the values from the current chunk it will be split by moving the top DSIZE/2 entries into a new chunk. The split operation is identical to the one performed during insertions, except that no key is inserted. All values but k are then copied into the next chunk, and the original is marked as a *zombie*. Lastly, down-pointers in the level above are redirected to unlink the *zombie*. The number of chunks in the level is incremented and decremented accordingly.

Copying the keys to the next chunk is performed in a manner similar to copying keys into a new chunk during a split. The order of operations is such that higher indexes are updated first, so that traversing teams (which give precedence to higher `tIds`) are not affected.

Care must be taken if k is in the last chunk in a level. A merge operation pushes values into the next chunk, which is impossible in this case. Thus entries are simply removed, even if this causes the chunk to be completely emptied. There can only be one such chunk in any level, and subsequent inserts and merge operations can add new values to it as necessary. The last chunk will never be marked a *zombie*, ensuring that all lateral traversals eventually reach a non-*zombie* chunk. If the last chunk in a level contains only the $-\infty$ key after the deletion then the chunk counter for that level is decremented to show that the level is empty.

The reader should note that all operations in GFSL were designed to be performed by a team in tandem, with only a few divergent `tId`-specific operations scattered throughout. The memory layout is such that every global memory access by a team is to memory-contingent locations. Thus we maximize memory coalescence and reduce divergence.

C. Some Words on Correctness

In this subsection we briefly mention a couple of major invariants used by our algorithm. One important promise is that a traversal will always reach the *enclosing* chunk of the key it is searching for (k) by taking only down and right steps. Our main concern when taking a step is that we never read a chunk to the right of k's *enclosing* chunk. Down-steps will never cause a bypass of k's *enclosing* chunk in the level below because keys can only be moved as part of split and merge operations, and then only to the right of their original chunk. A second important invariant is that the max field of a chunk can only decrease from the moment it is allocated, which is important in ensuring that teams taking lateral steps do not miss the enclosing chunk of a key. This means that once a key is placed in the data array of some chunk ch a larger key will never be inserted into any chunk to the left of ch. This continues to be true even if the key is later deleted from ch, or if ch becomes a *zombie*. Thus a lateral step will always reach either the *enclosing* chunk or a chunk to the left of the *enclosing*.

V. MEASUREMENTS/RESULTS

We evaluated GFSL compared to the skiplist algorithm ported to the GPU by Misra and Chaudhuri [7]. The code for their implementation is available online [24]. In the remainder of this section we refer to their implementation as “M&C”.

Both GFSL and M&C were evaluated on a GM204 GeForce GTX 970 (Maxwell architecture) GPU. We use the latest CUDA driver version 7.5 supporting compute capabilities 5.2. GTX 970 has 13 active streaming multiprocessors and a total of 1,664 cores. The device memory capacity is 4 GB GDDR5. The L2 Cache size is 1.75 MB. The core and memory clocks are 1050MHz and 1750MHz respectively. The operating system is 64-bit Ubuntu Server 14.04 with Linux kernel version 3.13.0-88-generic.

A. Experimental Setup

In this paper we observed four aspects that impact performance. The first is the structure size, which effects the traversal length and the amount of nodes that the GPU can hold in cache. The second is the percentage of updates and searches performed, as update operations are slower than searches. The third is GPU-specific configurations, such as the number of threads launched, their division into blocks, the number of operations performed by each team/thread, and, for GFSL, team/chunk size. The last is the value of p_{key} for M&C and p_{chunk} for GFSL. We choose to focus on the first two as they are more universal to all GPUs, while we optimized the last two to fit our current setup. While the rest of this chapter will be devoted to exploring the first two parameters, we will first discuss the latter.

In GFSL we use teams of 32 threads and chunks of size 256B with 32 8B key-value pairs. We set a limit on the

Table II
EFFECTS ON GFSL OF LIMITING WARPS LAUNCHED PER BLOCK

Warps per Block	8	16	24	32
Occupancy/ Theoretical	36.7%/ 37.5%	48.8%/ 50%	73%/ 75%	95.8%/ 100%
Registers	79	64	40	32
Active Blocks	3	2	2	2
Throughput (MOPS)[1]	58.9	65.7	62.5	52.9

[1] Throughput for operation mixture [10,10,80], range 1M

number of threads that can run in parallel, thus ensuring each thread receives more local resources, e.g. registers. Specifically, we launch 16 warps per block (512 threads) out of a possible 32. Under this limit GFSL launches 2 blocks per SM with 64 registers per thread, giving an occupancy of around 48.8% out of a theoretical 50% In this way we do not utilize the maximum possible parallelism supported by the hardware, but reach better results as there is less local memory "spillover".

Table II shows the effects on throughput, SM occupancy, register-per-thread allocation on GFSL, and active blocks per SM as a function of the number of warps launched per block. The throughput example shows that there is a tradeoff between the amount of concurrency (total threads launched) and the available local registers.

We tested M&C under several different configurations, varying the value of p_{key}, the number of warps, and the number of operations per thread. For the *Contains*-only test some configurations of M&C showed up to 24% better performance in low ranges than those shown in this paper. However, the *Contains*-only tests had unstable performance and very large confidence intervals. Thus, we chose the configuration that yielded the best results on average. M&C is configured to run 16 warps per block, with a single operation executed by each thread. This correlates to the best configuration described in the original paper.

The value of p_{key}/p_{chunk} influences both the number of layers traversed and the number of keys/nodes in each level. We tested different values for each, and used those that gave the best results. GFSL uses $p_{chunk} \approx 1$, (each chunk has one key on average leading to it from the level above). This effectively results in $p_{key} = 0.05$, as there are 20 entries per chunk on average. M&C uses $p_{key} = 0.5$.

We evaluate both skiplist implementations with several different operation mixtures. Mixtures are represented as tuples $[i,d,c]$ signifying a set of random operations with a probability of *i% Inserts*, *d% Deletes*, and *c% Contains*. The mixtures presented are [1,1,98], [5,5,90], [10,10,80] and [20,20,60], each evaluated by running 10M operations in varying key ranges between 10K and 100M. We also present benchmarks for each operation type (*Insert*, *Delete*, *Contains*) alone in the same key ranges. As above, *Contains* is tested with 10M operations. The number of operations in the *Insert* and *Delete* tests is equal to the key range, i.e. for a

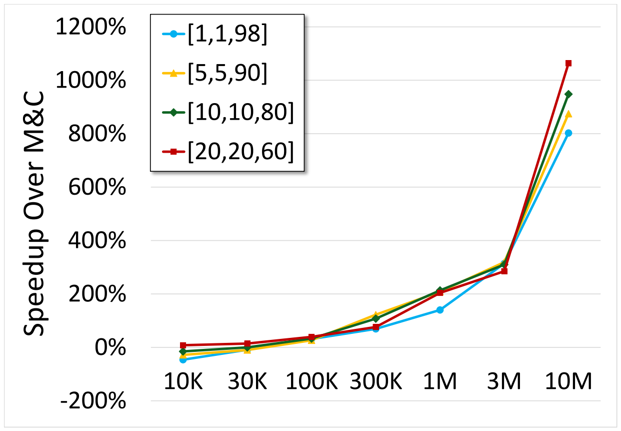


Figure 8. Ratio between GFSL and M&C as a function of the key range.

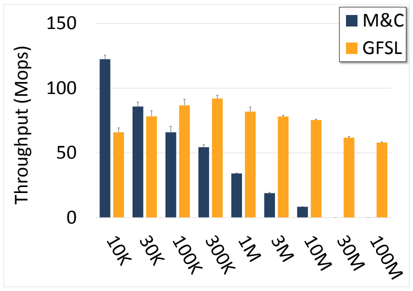
range of 100K keys, 100K operations were performed. This is in order not to oversaturate small structures.

The input to the CUDA test kernels for both implementations is an array of operations. Each entry in the array in GFSL consists of the operation type and a key. The array in M&C consists of an operation indication, key, and a value indicating level to which each key should be inserted. In both cases *Insert* operations use NULL as the value to be inserted. The operation type and keys for each entry are generated using uniform random functions, according to the configurations of the specific test. The initial structure on which the mixed-operation tests are performed contains a random set of keys, exactly half the size of the key range. Similarly, the initial structure for the *Contains*-only and *Delete*-only tests contains all of the keys in each range, inserted in a random order. The initial structure for the *Inserts*-only test is empty. Thus there is a direct correlation in our tests between the size of the range and the structures overall size. We run each experiment ten times and present the mean values along with 95% confidence intervals.

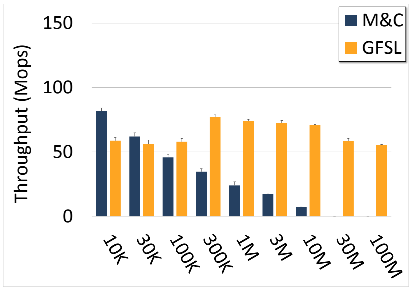
B. Performance Results

Fig. 8 shows the speedup of GFSL over M&C. GFSL is slower than M&C by up to 47% in the 10K range, up to 10% in the 30K range, then outperforms them by 27% to 1064% in the higher ranges. In Fig. 9 we present the actual throughput results of the tests. The figure shows that GFSL's peformance does not change drastically as the range increases, in contrast to M&C which melts down quickly as the range, and so the structure size, grows. This is the root cause of the rising ratio in the previous graph.

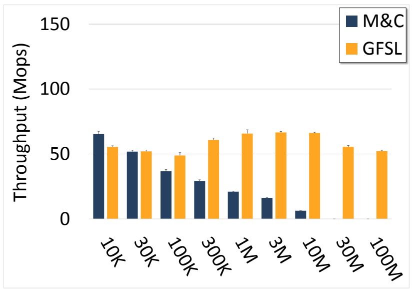
The main advantage of GFSL is the usage of coalesced reads, which optimizes accesses to the global memory. In the smaller range (10K), the entire structure fits into the L2 cache in both implementations, which significantly reduces the benefits of the coalesced reads as L2 access is much



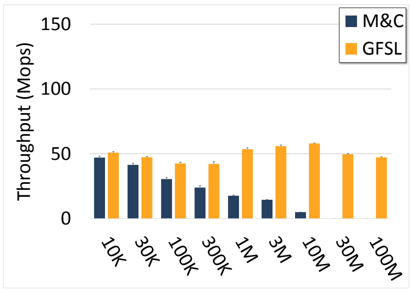
(a) [i,d,c]=[1,1,98]



(b) [i,d,c]=[5,5,90]

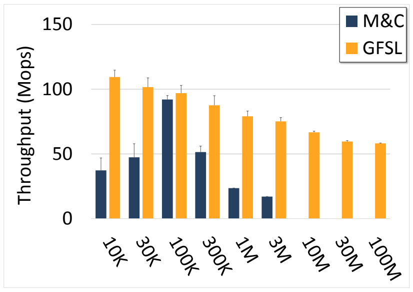


(c) [i,d,c]=[10,10,80]

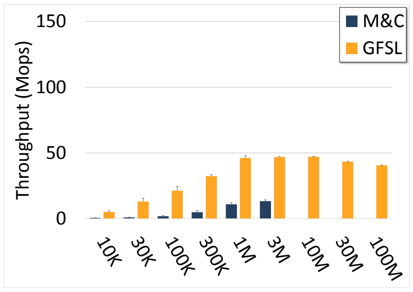


(d) [i,d,c]=[20,20,60]

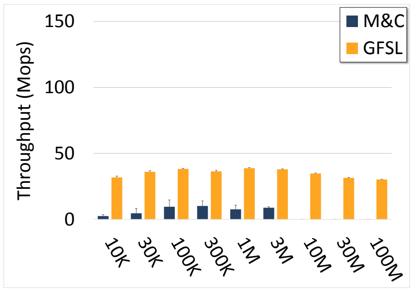
Figure 9. Throughput, in millions of operations per second, as a function of key range.



(a) 100% *Contains*



(b) 100% *Inserts*



(c) 100% *Deletes*

Figure 10. Throughput, in millions of operations per second, as a function of key range. Each graph shows the throughput of a single operation type.

faster than global memory access. However, in larger key ranges, M&C requires frequent uncoalesced accesses to the global memory that causes a sharp degradation in performance. GFSL does not suffer from this fast degradation. For example, comparing the key ranges 1M and 10M (a 10x larger structure) in the mixed-ops test, the performance of M&C is reduced by 68%-75%, whilst the performance of GFSL is reduced by up to 8%.

In addition to the key range, the performance is also impacted by the operation distribution. For the 10K range, M&C is faster than GFSL by 15%-47% when the percentage of *Contains* operations is high (Fig. 9a-9c), and slower by 8% when the percentage of *Inserts* and *Deletes* grows (Fig. 9d). The impact of the distribution is less than the impact of the key range, as GFSL's performance is closer to M&C's in the 30K range then quickly outperforms them in larger key ranges for all mixed distributions.

Looking at GFSL, we see a dip in performance in each of the mixed-ops tests. This dip occurs in small ranges when the number of update operations is small, and in larger ranges as the percentage of update operations grows (e.g. 300K for [20,20,60]). Smaller key ranges express a tradeoff between faster traversal and higher contention. Small structures allow faster traversals, both because more of the structure can reside in the cache and because fewer steps are required in traversals. However, when operations are generated from a smaller range of keys there is more chance for contention. The performance dip occurs when the benefit of small structure size cannot cover the loss from contention. As more updates are performed the dip occurs in larger key ranges, for which the structure is large enough not to benefit as much from faster traversals, but is small enough to still suffer from contention. This trend is reinforced in Fig. 10a, which shows the results of the *Contains* test. In this case there are no updates, thus no contention and no dips in performance.

M&C's implementation was measured up to the 10M range in the mixed-ops tests, and up to the 3M range in the single-op-type tests, as it runs out of memory for larger structures. In contrast, GFSL's compact layout and partial reuse of chunks allow it to run up to the range of 100M.

GFSL outperforms M&C for all single-op-type tests, as seen in Fig. 10. GFSL's *Contains* operation is faster than M&C by up to 4.4x in the large key ranges, and up to 2.9x in the low key ranges (Fig. 10a). M&C show surprisingly low performance in small key ranges in the *Contains* test, especially when considering the trends in the mixed-ops tests with few update operations; we were unable to determine the cause of the low performance. Fig. 10b and Fig. 10c show the performance of *Insert*-only and *Delete* only executions respectively. Both graphs show higher performance for GFSL in all ranges, between 3.5x-9.1x for *Insert* operations and between 3.5x-12.6x for *Deletes*.

VI. Related Work

While relatively little research has gone into designing general purpose data structures optimized for the GPU, some have been developed.

Hong et al. [25] showed that graph algorithms can be greatly accelerated on the GPU by designing a structure that emphasizes memory coalescing and warp-level cooperative execution. More recently, Zhang et al. [23] used similar techniques in their implementation of MegKV, an in-memory key-value storage system; in the context of a a GPU-friendly cuckoo hash table. MegaKV provided a speedup of 1.4-2.8 over the CPU implementation of the general algorithm.

Other hash tables have been designed and/or implemented on the GPU [26]–[30]. Bordawekar [27] proposed multi-level bounded linear probing, improving locality by using multiple levels of hash tables that reduce the number of lookups. Alcantara et al. [26] developed a cuckoo hashing scheme that achieves fast construction on the GPU and ensures lookup succeeds within at most 4 steps. Another cuckoo hashing scheme, [28], uses Collaborative Lanes, a method enabling threads in a warp to take on new tasks and so battle warp under-utilization.

Misra and Chaudhuri [7] tested the speedup of several known lock-free data structure algorithms ported to the GPU, in comparison with the CPU. Their results indicate that while a speedup is achieved on the GPU, increasing the dataset size and number of operations significantly reduces the GPU's advantage, especially in the case of more complex data structures such as skiplists and priority queues. Cederman et al. [8] performed similar experimentation on a variety of known lock-based and lock-free queue implementations, concluding that GPU-oriented optimization would benefit performance. In this work we show that a GPU-friendly design can perform significantly better.

Simpler data structures such as queues [31] and linked-lists [32] have been developed for the GPU. Some graph-based algorithms have also been sped up using GPU-optimized implementations [33]–[35].

Search trees geared towards graphics applications have also been GPU-optimized to good effect [36]–[38]. However, such structures typically distinguish between a construction phase in which elements are inserted, and a use phase in which elements are searched (but are never modified). They do not allow an intermix of these phases and so are not a good fit for general purpose applications.

Condensing data into contiguous areas of memory is a well-known technique for accelerating data structure operations in vector SIMD architectures. Several such structures have been designed such as binary search trees [39], b+-trees [40]–[42], and hash tables [43]. Sprenger et al. [44] designed a cache conscious skiplist with index levels in memory contiguous arrays and a linked list in the bottom level. The index levels are rebuilt periodically.

Braginsky and Petrank developed a locality-conscious linked list [45] and B+tree [46] for use in storage systems. A chunk based node design was proposed for the linked list and later used in the B+ tree implementation. As the cache-alignment requirement for efficient GPU programming can be compared to requirements for page-conscious systems the possibility of developing such structures to GPU programs is an interesting research question.

VII. Conclusion

We presented GFSL, a GPU-friendly algorithm for the skiplist data structure which utilizes chunked skiplist nodes and warp-cooperative functions to improve performance on the GPU. We demonstrated the importance of designing such specialized algorithms when attempting to execute non-streaming applications on a GPU by presenting a skiplist design that outperforms a straightforward porting of the CPU implementation to the GPU. We implemented our design on a GeForce GTX 970 Nvidia GPU (Maxwell architecture), and the results show a significant speedup of up to 11.6x over previous implementations for substantial key ranges. We believe that similar design considerations can be used to aid in efficient porting of other irregular-access concurrent data structures to the GPU environment.

References

[1] Nvidia, "CUDA C Programming Guide v7.5, september 2015. NVIDIA Developer Zone: website," 2015.

[2] OpenCL, "OpenCL 2.1 Reference Pages, The Khronos Group Inc.: website," 2015.

[3] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep Image: Scaling up Image Recognition," *arXiv preprint arXiv:1501.02876*, vol. 7, no. 8, 2015.

[4] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[6] Nvidia, "CUDA C Best Practice Guide v7.5, September 2015, NVIDIA Developer Zone: website," 2015.

[7] P. Misra and M. Chaudhuri, "Performance Evaluation of Concurrent Lock-Free Data Structures on GPUs," in *18th IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 53–60.

[8] D. Cederman, B. Chatterjee, and P. Tsigas, "Understanding the Performance of Concurrent Data Structures on Graphics Processors," in *European Conference on Parallel Processing*. Springer, 2012, pp. 883–894.

[9] RocksDB, "A Persistent Key-Value Store for Fast Storage Environments," http://rocksdb.org/, 2014.

[10] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[11] N. Shavit and I. Lotan, "Skiplist-Based Concurrent Priority Queues," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 2000, pp. 263–268.

[12] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," *arXiv preprint arXiv:1110.4623*, 2011.

[13] W. Pugh, "Skip Lists: a Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[14] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[15] W. Pugh, "Concurrent Maintenance of Skip Lists," Institute for Advanced Computer Science, Department of Computer Science, University of Maryland, Tech. Rep. CS-TR-2222.1, 1990.

[16] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A Provably Correct Scalable Concurrent Skip List," in *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[17] D. Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit, "ThreadScan: Automatic and Scalable Memory Reclamation," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit. - SPAA '15*. New York, New York, USA: ACM Press, 2015, pp. 123–132. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2755573.2755600

[18] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi, "Fast and Robust Memory Reclamation for Concurrent Data Structures," in *Proc. 28th ACM Symp. Parallelism Algorithms Archit. - SPAA '16*. New York, New York, USA: ACM Press, 2016, pp. 349–359. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2935764.2935790

[19] T. A. Brown, "Reclaiming Memory for Lock-Free Data Structures," in *Proc. 2015 ACM Symp. Princ. Distrib. Comput. - Pod. '15*. New York, New York, USA: ACM Press, 2015, pp. 261–270. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2767386.2767436http://dl.acm.org/citation.cfm?id=2767386.2767436

[20] N. Cohen and E. Petrank, "Efficient Memory Management for Lock-Free Data Structures with Optimistic Access," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit. - SPAA '15*. New York, New York, USA: ACM Press, 2015, pp. 254–263. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2755573.2755579

[21] N. Cohen and E. Petrank, "Automatic Memory Reclamation for Lock-Free Data Structures," in *Proc. 2015 ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl. - OOPSLA 2015*, vol. 50, no. 10. New York, New York, USA: ACM Press, 2015, pp. 260–279. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2814270.2814298

[22] D. Dice, M. Herlihy, and A. Kogan, "Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures," in *Proc. 2016 ACM SIGPLAN Int. Symp. Mem. Manag. - ISMM 2016*. New York, New York, USA: ACM Press, 2016, pp. 36–45. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2926697.2926699

[23] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[24] P. Misra and M. Chaudhuri, "Source code for lock-free data structure implementation (POSIX threads and CUDA)," in http://www.cse.iitk.ac.in/users/mainakc/lockfree.html, 2012.

[25] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 46, no. 8. ACM, 2011, pp. 267–276.

[26] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the gpu," *GPU Computing Gems*, vol. 2, pp. 39–53, 2011.

[27] R. Bordawekar, "Evaluation of parallel hashing techniques," *GTC*, 2014.

[28] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan, "Stadium Hashing: Scalable and Flexible Hashing on GPUs," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 63–74.

[29] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing." in *USENIX Annual Technical Conference*, 2016, pp. 281–294.

[30] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent parallel hashing," in *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6. ACM, 2011, p. 161.

[31] T. R. Scogland and W. Feng, "Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 63–74.

[32] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," in *Computer Graphics Forum*, vol. 29, no. 4. Wiley Online Library, 2010, pp. 1297–1304.

[33] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *International Conference on High-Performance Computing*. Springer, 2007, pp. 197–208.

[34] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[35] D. Merrill, M. Garland, and A. Grimshaw, "High-Performance and Scalable GPU Graph Traversal," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, p. 14, 2015.

[36] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-tree Construction on Graphics Hardware," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, p. 126, 2008.

[37] L. Luo, M. D. Wong, and L. Leong, "Parallel Implementation of R-Trees on the GPU," in *17th Asia and South Pacific Design Automation Conference*. IEEE, 2012, pp. 353–358.

[38] K. Zhou, M. Gong, X. Huang, and B. Guo, "Highly Parallel Surface Reconstruction," *Microsoft Research Asia*, 2008.

[39] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 339–350.

[40] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 795–806, 2011.

[41] S. Zeuch, F. Huber, and J.-c. Freytag, "Adapting tree structures for processing with simd instructions," 2014.

[42] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 145–156.

[43] K. A. Ross, "Efficient hash probes on modern processors," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 1297–1301.

[44] S. Sprenger, S. Zeuch, and U. Leser, "Cache-sensitive skip list: Efficient range queries on modern cpus," in *International Workshop on In-Memory Data Management and Analytics*. Springer, 2016, pp. 1–17.

[45] A. Braginsky and E. Petrank, "Locality-Conscious Lock-Free Linked Lists," in *International Conference on Distributed Computing and Networking*. Springer, 2011, pp. 107–118.

[46] A. Braginsky and E. Petrank, "A Lock-Free B+ Tree," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 58–67.