הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

A GPU-Friendly Skiplist Algorithm

Nurit Moscovici

A GPU-Friendly Skiplist Algorithm

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Nurit Moscovici

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5777 Haifa July 2017

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's MSc degree, the most up-to-date versions of which being:

Nurit Moscovici, Nachshon Cohen, and Erez Petrank. A gpu-friendly skiplist algorithm. In *International Conference on Parallel Architecture and Compilation Techniques (PACT), 2017*, 2017. (in press).

Nurit Moscovici, Nachshon Cohen, and Erez Petrank. Poster: A gpu-friendly skiplist algorithm. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 449–450. ACM, 2017.

The paper has been chosen as a contender for the best paper award at PACT 2017, which has not yet taken place as of the time of writing this thesis.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

We propose a design for a fine-grained lock-based skiplist optimized for execution on Graphics Processing Units (GPUs). GPUs have become increasingly popular in recent years as a platform for accelerating general purpose computations (GPGPU). GPUs are often used to accelerate streaming parallel computations, and it has been shown that highly data-intensive applications can achieve an order of magnitude speedup when run on a GPU. However, 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. Many efficient implementations of concurrent data structures have been designed and are widely used in parallel applications for the CPU. However, many of these algorithms do not fit with the specialized architectural requirements of the GPU, and may not scale well or even perform correctly. 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.

Abbreviations and Notations

GPU	:	Graphics Processing Unit
GPGPU	:	General Purpose Programming on the GPU
CPU	:	Central Processing Unit
GFSL	:	GPU-Friendly Skiplist
M&C	:	Skiplist implementation by Misra and Chaudhuri [MC12b]
JIT	:	Just In Time compilation
SPMD	:	Single Program Multiple Data
SIMT	:	Single Instruction Multiple Thread
SIMD	:	Single Instruction Multiple Data
SM	:	Streaming Multiprocessor
CUDA	:	Compute Unified Device Architecture, by Nvidia.
OpenCL	:	Open Computing Language, by Intel
B	:	Byte
GB	:	Gigabyte
L2	:	Level 2 Cache
$warp$	:	32 threads, the smallest unit scheduled by the SM
$half\text{-}warp$	:	Either the first or last 16 threads in a warp
$team$	:	A group of up to 32 threads that cooperate to perform GFSL operations
$chunk$	:	A node in GFSL
tId	:	Thread Id within a team (between 0 and ¡team size¿ - 1, inclusive)
N	:	The number of entries in a chunk
DATA	:	One of the first $N-2$ entries in a chunk, holding key-value pairs
NEXT	:	The entry in a chunk containing the pointer to the next chunk
LOCK	:	The entry in a chunk containing the lock
$split$	:	Operation for handling overfull chunks
$merge$	:	Operation for handling underfull chunks
$zombie$	:	A chunk logically, but not physically, removed from the structure
p_{key}	:	Probability that a key will be in level $i+1$ if it is in level i
p_{chunk}	:	Probability that a chunk in level i is represented by a key in level $i+1$
$[i, d, c]$	:	Operation mixture with i% inserts, d% deletes, and c% contains

Chapter 1

Introduction

In recent years, interest has surged in utilizing GPUs as a platform for accelerating general purpose programs (GPGPU). Today's GPUs are widely available at a low cost, and 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 [Nvi15b] and OpenCL [Ope15] over the past decade have opened GPUs for GPGPU programming without need for a background in computer graphics. GPUs are used today to accelerate applications in a wide variety of fields from deep learning [WYS+15] to database operations [BS10]. 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 [BNP12, Nvi15a]. 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 [HS12], attempts to port them directly to the GPU have shown that further GPU-specific optimizations are necessary [MC12b, CCT12]. Several GPU-based search structures geared toward graphics applications have been designed with good results [ZHWG08, LWL12, ZGHG08]. However, these structures typically distinguish between build and search phases, and do not allow for dynamic updates.

Relatively few dynamically updated concurrent data structures have been designed and optimized for the GPU. Some hash table designs, both based on linear probing [Bor14] and cuckoo hashing [ZWY+15, AVS+11, KBGB15], have been proposed in recent years. Of these, [KBGB15] and [ZWY+15] are dynamically updated. Simpler data structures such as queues [SF15] and linked-lists [YHGT10] have also been developed for the GPU.

The implementation of nonstreaming algorithms on GPUs is still in early stages, and 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 used in algorithmic design. Our intention is to support this direction by focusing on important data structures, in this case, the skiplist. Such structures are a natural basis for development of smarter applications. The target application would probably run entirely on a GPU kernel and would invoke skiplist operations as part of its execution.

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 [Roc14, Car13] and for other data structures such as priority queues [SL00]. 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.

We propose GFSL, a GPU-friendly design for a fine-grained lock-based skiplist. GFSL consists of linked lists of array-based nodes, called *chunks*, each of which contains several consecutive keys. Threads are divided into teams the size of a warp or smaller. Threads in a team access the skiplist chunks 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. GFSL benefits significantly from this design as it enables threads to cooperate during operation execution by concurrently handling a large amount of data with each execution step.

We compare GFSL to an implementation of a lock-free skiplist algorithm running on the GPU written by Misra and Chaudhuri [MC12b], 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.

Chapter 2

Preliminaries

2.1 GPU And The CUDA Programming Model

This work was designed and implemented in Nvidia's CUDA C++ programming model [Nvi15b]. CUDA programs employ a hetrogeneous model: serial, low-data-intensive elements are executed on the *host* CPU, which calls functions on the GPU *device* for highly parallel and data intensive computations. Communication between the host and the device is achieved by transferring large datasets between the host and device memory, a slow process that poses a significant bottleneck.

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 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.

2.2 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 [Nvi15a] important considerations for efficient programming

in the GPU environment.

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 [SO11]; 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.

Memory Coalescing A major consideration for improving performance is memory access optimizations [Nvi15a]: 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.

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.

Resource Management SMs contain a fixed-size register bank which is divided evenly amongst the threads according to the resource requirements of the kernel design. If more resources are needed by the kernel than are available there will be a costly "spillover" into a designated area of the global memory. Thus, access to local variables can potentially be as expensive as global access. Resource distribution can be optimized either by simplifying program code so that fewer resources are required by each thread,

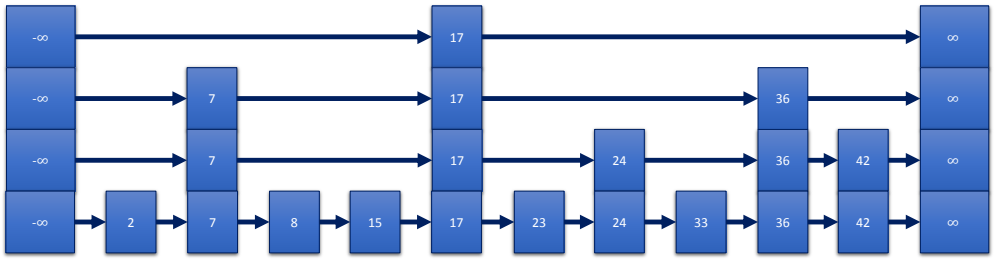


Figure 2.1: A classic skiplist structure

or by launching blocks that consist of fewer threads, enabling SMs to distribute more registers to each thread.

2.3 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 [Pug90b]. Many concurrent skiplist algorithms exist [HS12, Pug90a, HLLS06], though none have yet been designed with GPU-oriented optimizations.

A skiplist consists of layers of sorted linked lists, as in Figure 2.1. 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.

Chapter 3

Algorithm Overview

As discussed in Chapter 2, GPU algorithms are most efficient when performing coalesced memory accesses with low control flow divergence. We tune the classic skiplist structure to 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 Figure 3.1. 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. Chunks 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 Figure 3.2. 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

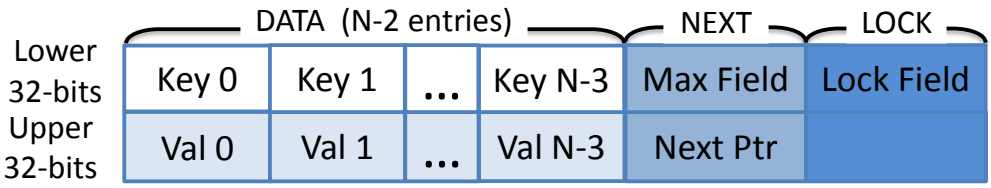


Figure 3.1: Format of a chunk of size N

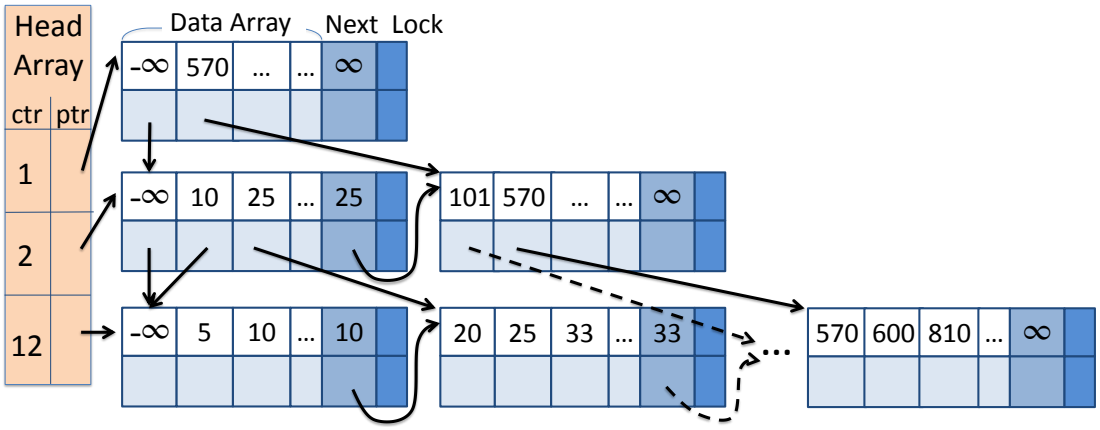


Figure 3.2: GFSL: A chunked skiplist

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 in GFSL 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 work 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 some chunk ch in level i will appear in level $i+1$

Chapter 4

Algorithm Details

4.1 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. These counters are used to keep track of the highest level currently in use 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. Chunks that are to be inserted into any place but the last in the level will have their max field updated to the maximum key contained within the data array before connection to the structure.

In this work chunk entries are of size 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 (3.5GB in the Maxwell architecture tested in this work) 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. [ZWY+15].

Some Definitions

Zombies: Chunks that have been removed by a merge operation but are still connected to the structure are called *zombies*. 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.

Table 4.1: 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

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 [ALMS15, BGHZ16, Bro15, CP15b, CP15a, DHK16], 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 [MC12b] by the fact that chunk entries from which keys have been removed can be reused as long as the chunk is not a *zombie*.

Enclosing Chunks: 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. Thus, a traversal searching for k terminates when it has found the *enclosing* chunk of k in the bottom level, regardless of whether k exists in the chunk.

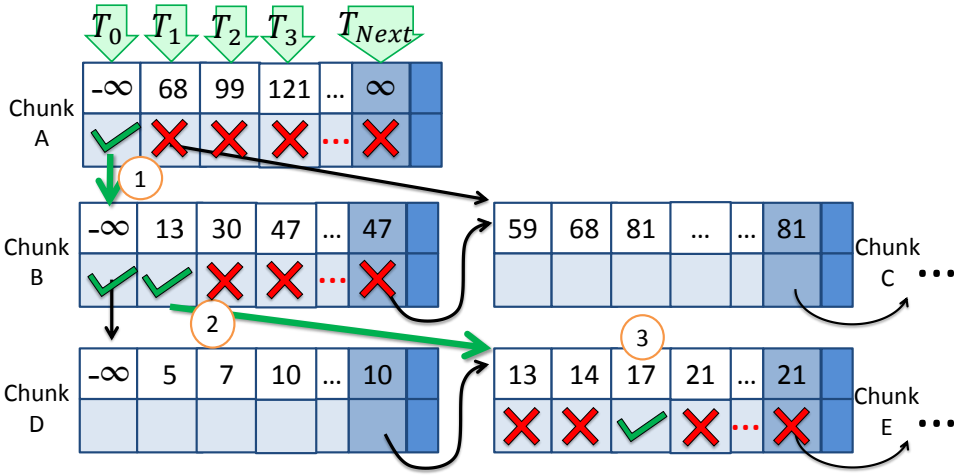
4.2 Data Structure Operations

In this section we present the algorithms for the *Insert*, *Delete*, and *Contains* operations in detail. Table 4.2 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.

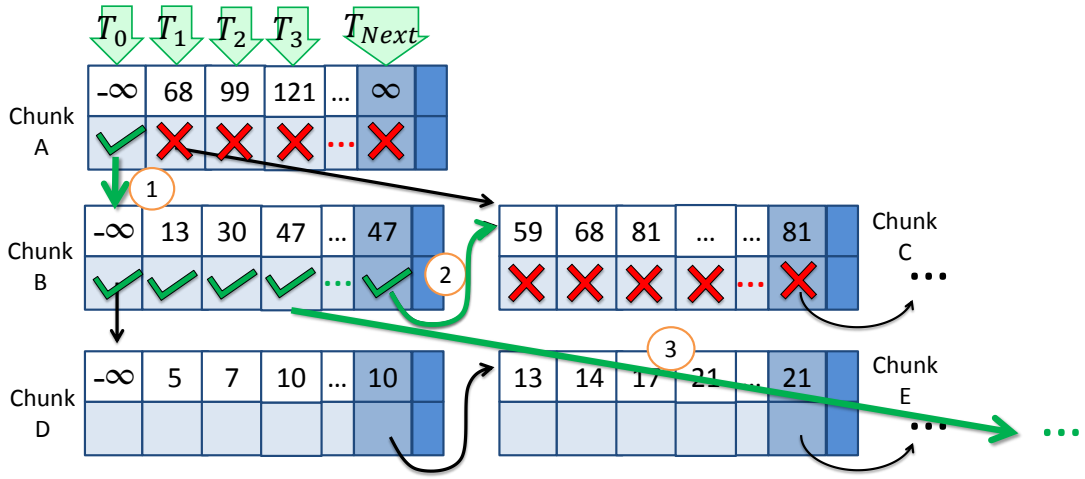
4.2.1 Contains

General Description

Contains are typically the most common operation called in programs using skiplists, and so it is vital that the traversal be as fast as possible. A *Contains* operation that



(a) A team performing *Contains*(17).



(b) A team performing *Contains*(52), entailing a backtrack.

Figure 4.1: Example of the *Contains* operation, pwrforming down-steps, lateral steps and backtracks

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 Figure 4.1 for an example of the *Contains* operation. In Figure 4.1a the team searches for key 17, beginning 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, indicating that a lateral step should be taken. The team uses `_ballot`, which receives the boolean result of this computation and returns a bitmap to each thread in which each bit represents the flag computed by the corresponding thread in the warp. The threads see that T_0 is the highest thread that returned *true*, and retrieve the pointer data 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.

Figure 4.1b showcases other possible types of steps in a traversal of GFSL with a search for key 52. As before, the team reaches Chunk B and each thread compares the key it read to 52. In this case, the `_ballot` shows that T_{NEXT} is the highest thread returning *true*, meaning that 52 is greater than all keys in the chunk. The team takes a lateral step into Chunk C, and again compares the keys found to 52. However, the `_ballot` now shows that no thread returned *true*, signifying that 52 is smaller than all keys in this chunk. In this case, the proper key to use for a down step would be key 47 in Chunk B, as it is the greatest key in the level that is less than 52. The team backtracks to Chunk B and steps down into the bottom level. There it will take lateral steps until the *enclosing* chunk of 52 is found.

Implementation Details

Algorithm 4.1 shows the *Contains* operation, which calls two main functions. The `searchDown` function, described in Algorithm 4.2, 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.

In each iteration the team reads a chunk from memory then uses the cooperative function `getTidForNextStep` described below and shown in Algorithm 4.3 to decide what the next step should be. There are three possibilities for the next step: a lateral

Algorithm 4.1 *Contains*

```
1  bool contains(K k)
2  {
3      CHK* pCurr = searchDown(k)
4      return searchLateral(k, pCurr)
5  }
```

step in the same level, a step down to the lower level, or a backtrack through the previous chunk in the same level.

As demonstrated in the example above, down steps, shown in Algorithm 4.2, Lines 19-23, occur when k is not greater than the maximum key in the chunk. Likewise, lateral steps (Lines 15-18) occur when k is greater than the maximum key in the chunk. In both cases, the cooperative function `getPtrFromTid` 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 (Lines 24-28, 33-38). In this case the team must step down using the maximum key in the previous chunk. 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 16).

The helper functions called by the `searchDown` algorithm are all cooperative. We consider `getTidForNextStep` and `getPtrFromTid` as examples of such functions. These functions showcase the usage of `_shfl` and `_ballot`, the main intra-warp operations used in this work. Other cooperative functions described in the rest of this chapter 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 `getTidForNextStep`, shown in Algorithm 4.3, we see usage example for the _ballot operation. Each thread simultaneously calculates a boolean value dependent on its `tId`, k, and the key it read from the chunk (Lines 3-4). The threads then call `_ballot` simultaneously (Line 6) to receive the results of this calculation for each thread. The NEXT thread passes a *true* 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, which is 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,

Algorithm 4.2 *SearchDown*

```
1   CHK* 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
14          int stepTid = getTidForNextStep(k, currKv)
15          if (stepTid == NEXT) { //lateral step
16              prevKv = currKv
17              pCurr = getPtrFromTid(NEXT, currKv)
18          }
19          else if (stepTid != NONE) { //down step
20              height --
21              prevKv = null
22              pCurr = getPtrFromTid(stepTid, currKv)
23          }
24          else { //backtrack
25              if(prevKv == null) goto search
26              height --
27              pCurr = backTrack(prevKv, k)
28          }
29      }
30      return pCurr
31  }
32
33  CHK* backTrack(KV& prevKv, K k){
34      int stepTid = getTidOfDownStep(k, prevKv)
35      CHK* pNextStep = getPtrFromTid(stepTid, prevKv)
36      prevKv = null
37      return pNextStep
38  }
```

Algorithm 4.3 *Examples of Cooperative Helper Functions*

```
1
2   void getTidForNextStep(K k, KV currKv){
3       bool elem = (tId < DSIZE) && (currKv.key <= k)
4       bool next = (tId == NEXT) && (currKv.key < k)
5
6       uint bal = __ballot(next || elem)
7       if (bal == 0)
8           return NONE
9       return 32 − clz(bal) − 1
10  }
11
12  CHK* getPtrFromTid(int winningTid, KV kv){
13      return __shfl(kv.v, winningTid) //take kv.v value from thread with given tId
14  }
```

signifying that a backtrack must be executed.

`getPtrFromTid`, in Line 12 of Algorithm 4.3, shows a usage example for the `_shfl` operation. Each thread specifies the value field it is interested in receiving, and the thread from which it wishes to receive the value. In this case, each thread takes the value field of the KV pair held by the thread with $tId == winningTid$.

Searching along the bottom level is performed by the `searchLateral` function presented in Algorithm 4.4. 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, rather than less-than-or-equal (Line 4). The NEXT thread continues to check whether a lateral step is necessary. The team calls the cooperative function `isTidWithEqualKey` 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.

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 Figure 4.1a 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

Algorithm 4.4 *SearchLateral*

```
1   bool searchLateral(K k, CHK* pCurr){
2       do {
3           KV currKv = pCurr->read(tId)
4           int foundTid = isTidWithEqualKey(k, currKv)
5
6           if (foundTid == NEXT || isZombie(currKv)) {
7               foundTid = NEXT
8               pCurr = getPtrFromTid(NEXT)
9           }
10
11      } while(foundTid == NEXT)
12
13      return foundTid != NONE
14  }
```

enough information to continue, and so the traversal is restarted.

In more general terms, a restart occurs when a down-step is taken using a pointer associated with some key, $kDown$, which was concurrently deleted by another team. If the removal of $kDown$ causes the team to read a chunk in the lower level in which all the keys are greater than k then the team will decide to backtrack, despite the fact that the `prevKV` field was set to null after the down step (Algorithm 4.2, Line 21). 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*).

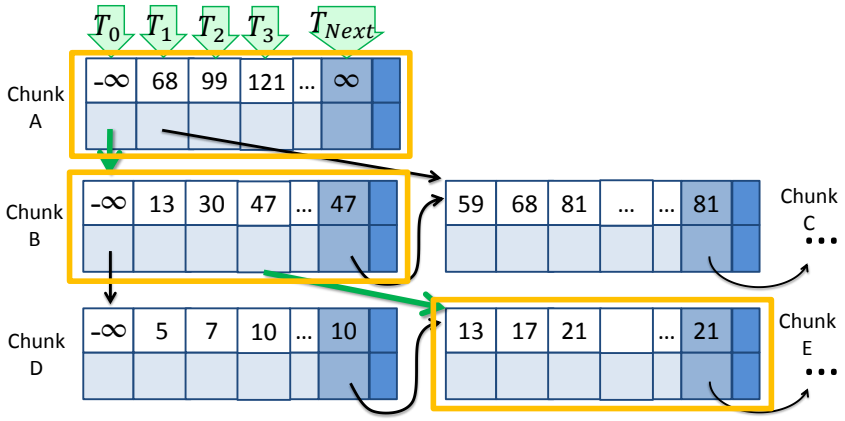
4.2.2 Insert

General Description

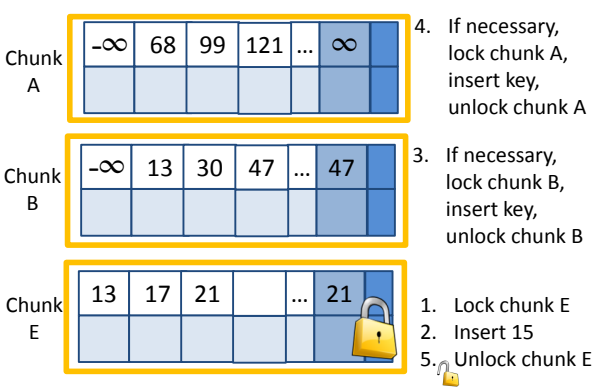
The *Insert* function receives $<k, v>$, the key-value pair to be inserted, and searches the structure for k. The insertion is executed only if k is not already in the structure, and performed bottom-up. If insertion causes a chunk overflow a split operation will occur and a new chunk will be added to 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}.

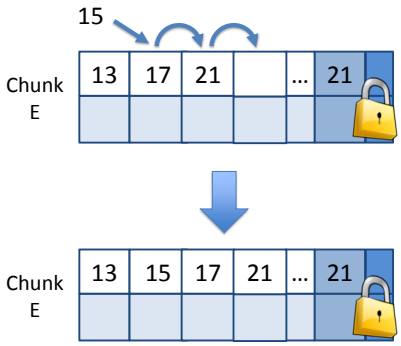
Consider Figure 4.2 as a simple example of an *Insert* operation, inserting key 15.



(a) Path taken during traversal to find the *enclosing* chunk of key 15. The path contains only one chunk per level – the chunk through which the down-step was taken



(b) General order of operations when inserting to multiple levels. The chunk on the bottom level remains locked for the duration. If it is necessary to insert into higher level chunks, lock-insert-unlock to reduce contention.



(c) Actual insertion of key 15 into chunk E. Keys greater than 15 are shifted to the right to preserve sorting. The insertion is performed cooperatively by the team.

Figure 4.2: Example of the *Insert* operation, inserting key 15

The team first traverses the structure to find the *enclosing* chunk of 15. Traversal is similar to that of the *Contains* operation, except that in this case the traversal *path* is recorded, as seen in Figure 4.2a. The path is used as a starting point for insertion into each level - the *enclosing* chunk must either be in the path or reachable from it. The team reaches Chunk E and locks it, then inserts key 15 while ensuring the sorted order remains, as shown in Figure 4.2c. If Chunk E is full it will be split, and a key will be inserted into the level above at a probability of p_{chunk} (in this case into Chunk B). If a split should then occur in Chunk B, the process will repeat itself in the level above (Chunk A). The lock on Chunk E is only released when all inserts to all levels have completed. If no split occurs, Chunk E will be unlocked immediately after the insertion of key 15. Figure 4.2b illustrates the order of locks and inserts in the various levels of the structure.

Implementation Details

The *Insert* function, presented in Algorithm 4.5, begins by searching for k using the `searchSlow` function (Algorithm 4.6), and returns *false* if k already exists in the structure. `searchSlow` performs the same traversal as *Contains*, with two main differences: firstly, `searchSlow` saves and returns the traversal path (Lines 4, 28-29, 36-37 in Algorithm 4.6). 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. 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 (Lines 10-20, 42). The redirection is performed lazily by calling try-lock on the previous chunk. If the lock fails the team continues without updating. If the *zombie* was the first chunk in the level, the head array will be updated accordingly. Update of down-pointers is discussed below.

One would expect a path to be an array of pointers to nodes in each level. Indeed, in the classic skiplist algorithm a traversing thread saves the search path in a local array. However, local arrays are costly in CUDA in terms of resources as they are often stored in the "spillover" area of global memory. 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, which hold an average of 20 keys, allow for around 20^{32} keys. Both are far beyond the global memory capabilities both in current GPUs and those in the foreseeable future.

If k was not found, `insertToLevel` (Algorithm 4.5, Lines 28-46) 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 8, 17, and 43). `insertToLevel` will return *false* if k was concurrently added by another team before the lock was caught. In Algorithm 4.5, Lines15-22, 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 12 and 19).

`insertToLevel` calculates the number of empty entries in the data array (Line 35). 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.

Algorithm 4.5 *Insert*

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

Algorithm 4.6 *SearchSlow*

```
1   <bool, CHK*> searchSlow(K k){
2   search:
3       KV prevKv = null
4       CHK* path = headPtrAtHeight(tId)
5       int height = getHeight()
6       CHK* pCurr = firstChunkAtLevel(height)
7
8       while(height>0) {
9           KV currKv = pCurr->read(tId)
10          if (isZombie(currKv)) {
11              CHK* firstNonZombie = findFirsNonZombie(currKv)
12
13              if (prevKv != null)
14                  redirectToRemoveZombie(prevKv, firstNonZombie, currKv)
15              else if (isFirstInLevel(pCurr, height)){
16                  updateHeadArray(height, firstNonZombie, pCurr)
17
18              pCurr = firstNonZombie
19              currKv = pCurr->read(tId)
20          }
21
22          int stepTid = getTidForNextStep(k, currKv)
23          if (stepTid == NEXT) { //lateral step
24              prevKv = currKv
25              pCurr = getPtrFromTid(NEXT, currKv)
26          }
27          else if (stepTid != NONE) { //down step
28              if (tId == height)
29                  path = pCurr
30              height --
31              prevKv = null
32              pCurr = getPtrFromTid(stepTid, currKv)
33          }
34          else { //backtrack
35              if(prevKv == null) goto search
36              if (tId == height)
37                  path = &prevKv
38              height --
39              pCurr = backTrack(prevKv, k)
40          }
41      }
42      return <findLateralWithZombieRedirect(k,pCurr), path>
43  }
```

Algorithm 4.7 *ExecuteInsert*

```
1   void executeInsert(CHK* pEnc, KV encKv, K k, V v){
2       KV insertKv = getChunkValFromLeftNeighbor(encKv)
3       uint insertIdx = getInsertionIdx(insertKv, k)
4       if (tId == insertIdx)
5           insertKv = pair(k,v)
6
7       for(int i = DSIZE−1; i >= insertIdx; i −−){
8           if ((insertKv.key != EMPTY) && (tId == i))
9               pEnc−>AtomicWrite(tId, insertKv)
10      }
11  }
```

`executeInsert` (Algorithm 4.7) inserts $<k, v>$ while ensuring the chunk remains sorted. In Line 2 each thread takes the key-value pair from the previous thread in the team using a cooperative function. Then, in Line 3 the insertion index for $<k, v>$ in the sorted data array is determined in another cooperative function. In Lines 7-10 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 Figure 4.3. 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 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.

Locking a Chunk

`findAndLockEnclosing` (Algorithm 4.8) is a spin-lock that performs a lateral search in order to ensure that the chunk being locked *encloses* k. If the current chunk does not *enclose* k (or is a *zombie*) the team will read the next chunk, as seen in Lines 5-8. Otherwise the function checks whether the chunk is unlocked before the LOCK thread attempts to lock it using CAS (Lines 10-13). 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. This sequence is repeated until the *enclosing* chunk is successfully locked.

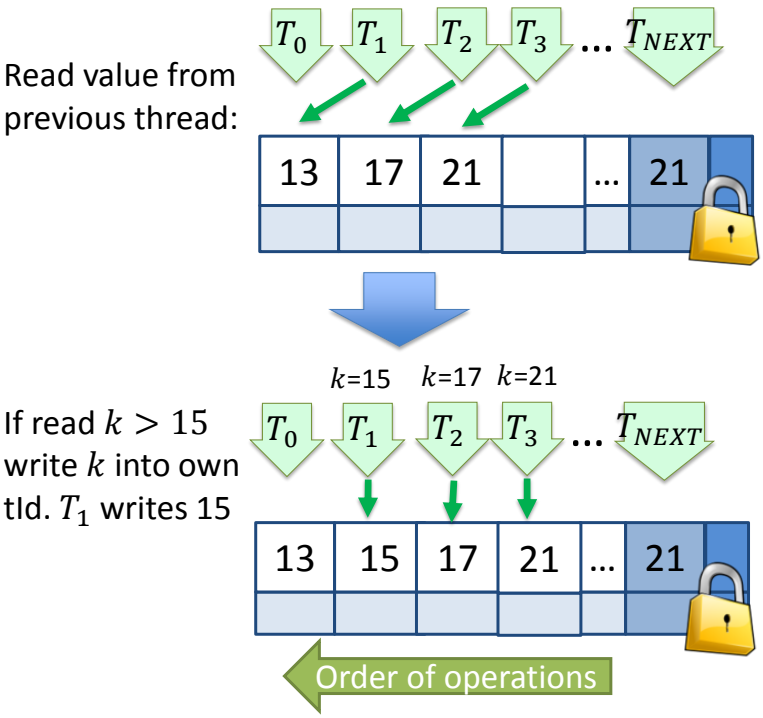


Figure 4.3: Inserting key 15 into a chunk without a split. Each thread reads the entry to its left, and if it is greater than 15 copies it into its own entry. Order of copying is from right to left.

Split

If the chunk is already full, the team calls `splitInsert` (Algorithm 4.9) to perform a split as shown in Figure 4.4. The `preSplit` function (Algorithm 4.9 Lines 16-21) locks the next chunk, removing *zombies* if they are encountered (Line 17), then allocates a new chunk which is initialized to point to the next chunk.

`splitCopy` (Algorithm 4.9 Lines 23-34) is then called to copy the top DSIZE/2 values to the new chunk (Lines 27-28). 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 (Line30). The team can then atomically write an empty value to each of the entries in the old chunk whose values were copied to the new (Line 32). Again, we rely on the fact that traversals give precedence to higher `tId`s to argue that a concurrent traversal will not be adversely affected. The updated max field ensures the NEXT thread's flag in the result of a `_ballot` operation is considered before those of DATA threads whose entries have not yet been emptied.

The split continues at Line 4 of Algorithm4.9, 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.

Algorithm 4.8 *FindAndLockEnclosing*

```
1   CHK* findAndLockEnclosing(KT key, CHK* ch){
2   start:
3       KV kv = ch->read(tId)
4
5       if (chunkNotEnclosing(kv, key)) {
6           ch = getPtrFromTid(NEXT)
7           goto start
8       }
9
10      if (isChunkLocked(kv))
11          goto start
12
13      if (!LockChunkWithCAS(ch, kv))
14          goto start
15
16      kv = ch->read(tId)
17      if (chunkNotEnclosing(kv, key)) {
18          unlockChunk(kv)
19          ch = getPtrFromTid(NEXT)
20          goto start
21      }
22
23      return ch
24  }
```

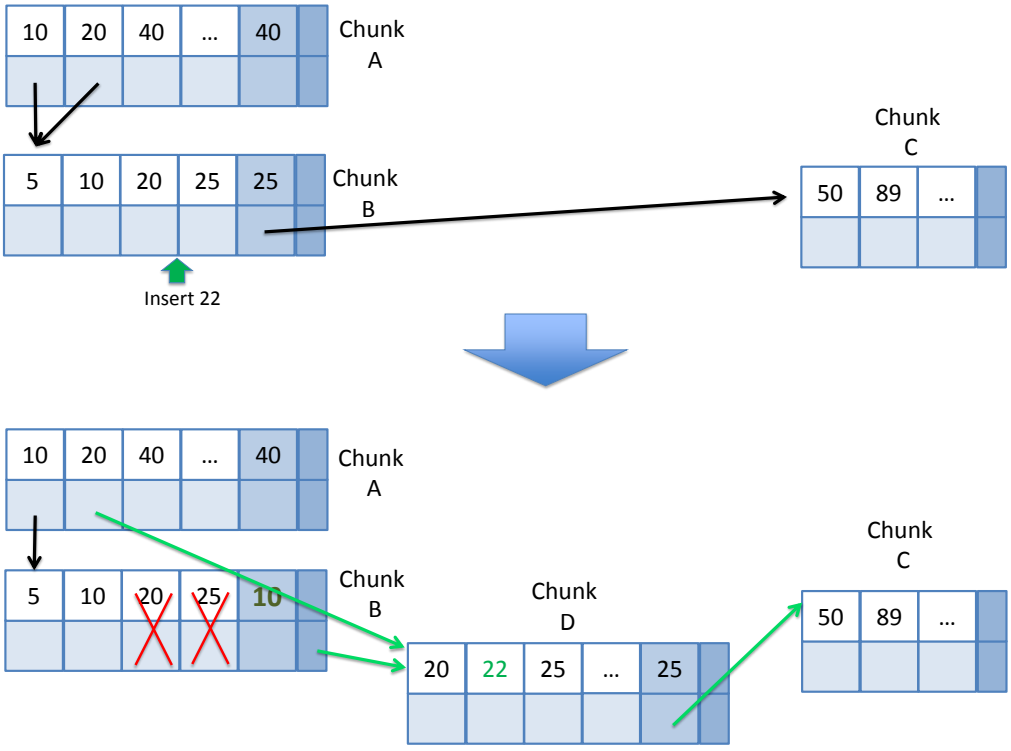


Figure 4.4: Insertion of key 22 causes a split. Keys 20 and 25 are moved from chunk B to D (the new chunk). B's next pointer and key 20's down-pointer in A are redirected to D.

Algorithm 4.9 *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
11      k = keyForNextLevel(k, pInsert, pNew, pSplit, level)
12      updateDownPtrs(level, splitKv, pNew)
13      return <pInsert, k>
14  }
15
16  CHK* preSplit(CHK* pSplit){
17      CHK* pNext = lockNextChunk(pSplit)
18      CHK* pNew = alloc()
19      updateNextField(pNew, pNext)
20      return pNew
21  }
22
23  KV splitCopy(Chk* pSplit, CHK* pNew){
24      KV splitKv = pSplit->read(tId)
25      K thresh = getKeyFromTid(splitKv.key, DSIZE/2-1)
26
27      if (splitKv.key > thresh)
28          copyToNewChunk(pNew, splitKv)
29      if (tId == NEXT)
30          updateNextField(pSplit, pNew)
31
32      setMovedValsEmpty(splitKv)
33      return splitKv
34  }
```

Algorithm 4.10 *UpdateDownPtrs*

```
1   void updateDownPtrs(int level, KT mKey, KT startTid,
2                       int numMoved, CHK* lowerMovedCh){
3       KT k = _shfl(mKey, startTid)
4       CHK* upperCh = searchDownToLevel(level+1, k)
5       int count = 0
6       while(count < numMoved) {
7           if (findLateral(k, upperCh)){
8               upperCh = findAndLockEnclosing(k, upperCh)
9
10              if (findLateral(k, lowerMovedCh)) //update chunk in upper level
11                  updateDownPtr(k, upperCh, lowerMovedCh)
12              unlockChunk(upperCh)
13          }
14          count++
15          key = _shfl(mKey + count, startTid)
16      }
17      return
18  }
```

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 11, 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 the locked chunk.

Updating Down Pointers

Finally, the team updates the down-pointers in level $i+1$ to reflect the changes in level i (Line 12), by searching level $i+1$ for the range of moved keys, then locking affected chunks and atomically updating relevant down-pointers. In The example in Figure 4.4, key 20 was moved in the split of Chunk B, causing its down pointer in Chunk A to be updated to point to Chunk D, which is now the chunk *enclosing* key 20. In the time between key 20 being copied to Chunk D and the pointer from Chunk A being updated it continues to point to Chunk B. This is legal in terms of traversal as Chunk D can be reached from Chunk B using lateral traversal.

`updateDownPtrs` (Aglorithm 4.10) is called after any change to the structure that causes keys to be moved between chunks in level i. The team attempts to fix pointers from level $i+1$ associated with keys that were movedin level i, either as a result of a split or a merge. The team searches for the minimum key moved in level $i+1$ (Line 4) using a function identical to `searchDown` except that it searches until level i and not

level 0. Each thread in the team holds one of the moved keys in its $mKey$ field. For every two threads T_m, T_n s.t. $m < n$ the key held by T_m is smaller than the key held by T_n. The team takes each such key in turn, from the lowest to the highest (Lines 3 and 15), and searches level $i+1$ for that key (Line 7). If the key is found the team locks the chunk in level $i+1$ (Line 8), searches for the *enclosing* chunk in level i(Line 10), and updates the pointer from $i+1$ (Line 11) before unlocking the chunk in level $i+1$.

4.2.3 Delete

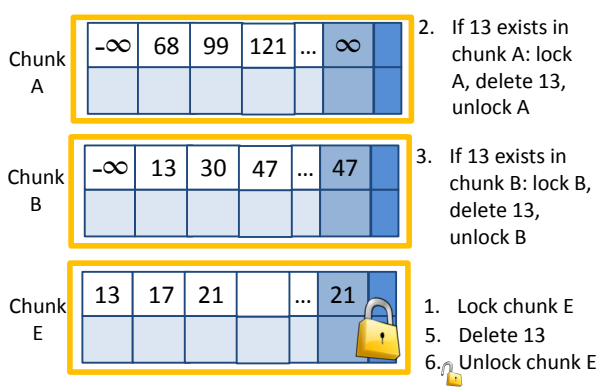
General Description

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 is found to exist 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 levels that are currently in use from the top down, removing k from 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.

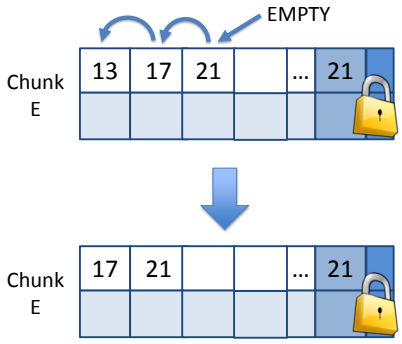
If the chunk from which a key is deleted crosses some minimum threshhold, a merge occurs. The values from the underfull chunk are moved to the next chunk to the right, causing a split of that chunk if necessary. The old chunk is marked as a *zombie*, and the team redirects pointers from the level above to reflect the changes. Copying keys during a merge is performed in such a way as to ensure that the chunk remains sorted, and that concurrent traversals will not be adversely effected.

Figure 4.5 shows an example of a *Delete* operation of key 13 which ends in a merge. As in *Insert*, the team first traverses the structure to find the path to the *enclosing* chunk of 13 in the bottom level and checks containment. The team locks chunk E in the bottom level, and ensures key 13 still exists there. The team then searches for key 13 in each level starting from the chunk saved as a part of the path, and performs a sequence of lock-delete-unlock operations as illustrated in Figure 4.5a. Chunk A does not contain key 13, and so is skipped. In the next iteration Chunk B is found to contain key 13 and is locked. The key is then deleted and Chunk B is unlocked. Finally, key 13 is deleted from Chunk E and the bottom level lock is released. The physical removal of the key is performed by shifting larger keys one entry to the left, as shown in Figure 4.5b;

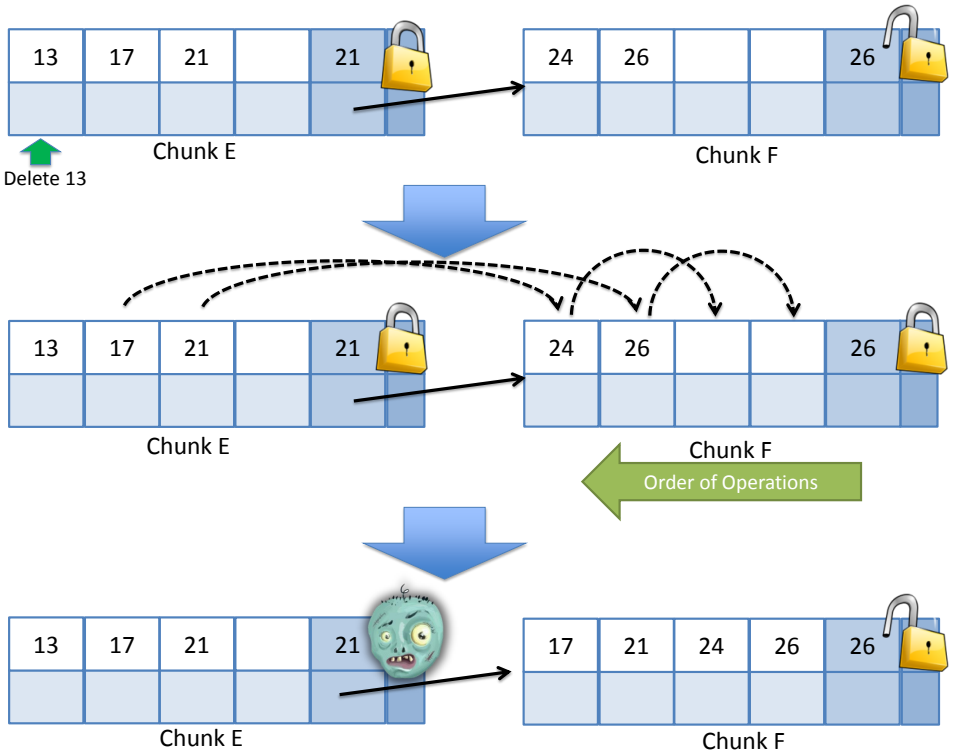
In Figure 4.5c we present the case in which removal of key 13 causes Chunk E to be merged. The next chunk, F, is locked, and keys 17 and 21 are copied into Chunk F. Keys 24 and 26 from Chunk F are moved to the end of the chunk in order to make room. This set of copy operations is performed from right to left in order not to temporarily overwrite any existing keys, and in order not to adversely effect concurrent traversals. Finally, Chunk F is unlocked, and the lock field of Chunk E is marked with a *zombie* value. The team will then traverse the structure to redirect pointers from the level above,



(a) General order of operations when deleting from multiple levels. The chunk on the bottom level remains locked for the duration. If the key exists in upper levels into higher level chunks, lock-delete-unlock to reduce contention. Delete from bottom chunk last.



(b) Actual removal of key 13 from chunk E. Keys greater than 13 are shifted to the right to preserve sorting. The insertion is performed cooperatively by the team.



(c) Removal of key 13 caused a merge in chunk E. All values but 13 are moved from E to F while ensuring F remains sorted. Finally, E is marked as a *zombie*

Figure 4.5: Example of the *Delete* operation. Key 13 is removed from the structure, causing a merge to occur.

thus removing down-pointers to the zombie. The pointer from the chunk preceding E in the structure will be lazily updated by some other operation. Note that key 13 is not actually physically removed from Chunk E. As E is now a zombie, it will be ignored by

Algorithm 4.11 *Delete*

```
1   bool delete(K k){
2       CHK* path
3       bool found = searchSlow(k, path)
4       if (!found)
5           return false
6
7       CHK* pBottom = getPathFromTid(0)
8       pBottom = findAndLockEnclosing(pBottom, k)
9       if (!chunkContains(pBottom, k))
10          return false
11
12      int height = rereadHeightAndUpdatePath(path)
13      for (int i = height; i > 0; i--) {
14          CHK* pEnclose = getPathFromTid(i)
15          if (!searchLateral(k, pEnclose))
16              continue
17
18          pEnclose = findAndLockEnclosing(pEnclose, k)
19          removeFromChunk(k, pEnclose, i)
20      }
21
22      removeFromChunk(k, pBottom, 0)
23      return true
24  }
```

all future operations, effectively removing key 13 from the structure.

Implementation Details

The *Delete* operation, shown in Algorithm 4.11, receives a key to be deleted (k) and begins by searching the structure. As in the *Insert* function, `SearchSlow` is called in Line 2 to find k and the traversal path to k. If k is not found the algorithm returns *false*. Otherwise the *enclosing* chunk of k in the bottom level is locked and containment of k is confirmed. The path is updated according to the current structure height, so as not to miss new levels that may have been added since the path was found.

In Lines 13-20 of Algorithm 4.11 the team iterates over all levels in the structure searching for and deleting k where found. In each level i, `searchLateral` is called to determine whether k exists in the level, and the *enclosing* chunk is locked only if k was found (Lines 15-18). If k was found not to exist in level i the team will continue to level $i-1$. Checking containment before the lock significantly reduces contention on the higher and less populated levels of the skiplist when there are a large number of concurrent calls to *Delete*. There is no need to recheck the containment of k after locking an upper level, as the lock on the bottom level ensures that no other team is concurrently updating k.

If k exists in level i `removeFromChunk` is called to remove it, performing a merge if necessary and unlocking any non-*zombie* chunks effected by the removal. Only in Line 22, after k has been removed from all upper levels, does the team delete it from the bottom level, and thus from the structure itself. The bottom chunk is only unlocked when the *Delete* operation concludes.

Algorithm 4.12 shows the `removeFromChunk` function. Removing k from a chunk in any level i is divided into three cases:

- k can be removed without performing a merge (Lines4-7)
- A merge is required (Lines 15-27)
- k is situated in the final chunk in level i (Lines 10-13)

A merge is deemed necessary if removing k will cause the number of nonempty entries in the data array to cross a predetermined minimum threshold (DSIZE/3 in this work).

Delete With No Merge If no merge is required the team calls `executeRemoveNoMerge`. `executeRemoveNoMerge` removes k in a manner similar to `executeInsert`, though in the opposite direction, as illustrated in Figure 4.6. Each thread reads the key-value pair corresponding to its own `tId` from the chunk. DATA threads with `tId`s equal to or higher than k's index atomically write their value into the entry to the left of 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, if the chunk was full before the removal of k the NEXT thread writes the EMPTY key into the last entry in the DATA array, as there is no DATA thread to the right of this entry to empty it.

Delete With Merge 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 (Line 9). 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 (Lines 16-19). The split operation is identical to the one performed during insertions, except that no key is inserted.

`executeRemoveMerge` is called to perform the merge operation by copying all values but k into the next chunk as illustrated in Figurefig:merge. The order of operations when copying keys to the next chunk is such that higher indexes are updated first, so

Algorithm 4.12 *RemoveFromChunk*

```
1   void removeFromChunk(K k, CHK* pEnc, int level){
2       KV encKv = pEnc->read(tId)
3       int count = numNonEmpty(encKv)
4       if (count > DSIZE/3) { // no merge required
5           executeRemoveNoMerge(encKv, pEnc, k)
6           unlockChunk(pEnc)
7       }
8       else { //merge is needed.
9           CHK* pNext = lockNextChunk(pEnc)
10          if(pNext == NULL) { // don't merge last chunk in level
11              removeFromLastChunk(k, pEnc, encKv)
12              return
13          }
14
15          KV nextKv = pNext->read(tId)
16          if (numNonEmpty(nextKv) + count - 1 > DSIZE) {
17              splitRemove(pNext, level)
18              incrementNumChunksAtLevel(level)
19          }
20
21          executeRemoveMerge(encKv, pEnc, nextKv, pNext, k)
22          markAsZombie(pEnc)
23
24          decrementChunksInLevel(level)
25          unlockChunk(pNext)
26          unlockChunk(pEnc)
27          updateDownPtrs(level, encKv, pNext)
28      }
29      return
30  }
```

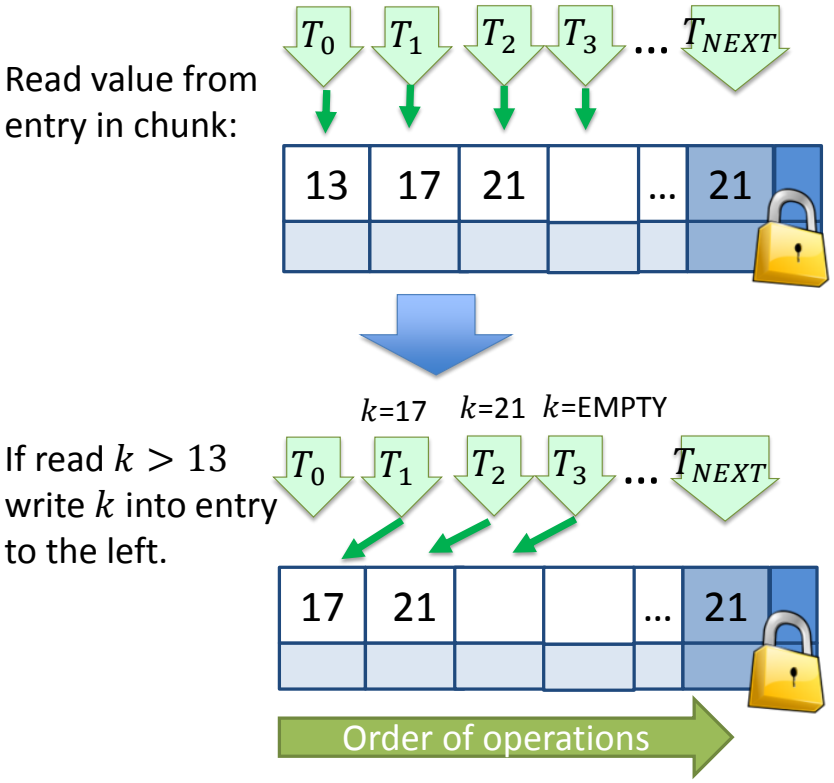


Figure 4.6: Deleting key 13 from a chunk. All keys greater than 13 are moved one entry to the left.

that traversing teams (which give precedence to higher `tId`s) are not affected. Copying the keys to the next chunk is performed by calling a series of `_shfl` operations after which each DATA thread holds the value that will appear in the next chunk after the merge.

The general idea is that the keys from the merged chunk migrate to the lower indexes of the next chunk, while the original entries in the next chunk are moved to the right to make space. The next chunk is updated by atomically writing each of the new values serially in descending order of `tId`. This may temporarily cause the next chunk to be unsorted, or even to contain a mixture of EMPTY and non-EMPTY entries.

Consider the merge in Figure 4.5c: keys 17 and 21 are copied from Chunk E to Chunk F. This causes the original keys from Chunk F, keys 24 and 26, to be moved to make room for the new keys. The order of operations is from right to left, so key 26 is the first to be copied, and is placed in the last empty space in F. Once this move has been completed, the second-to-last entry in F remains EMPTY, and both the second and last entries in F contain key 26. The traversal handles this as before by giving precedence to *true* values computed by higher *tId*s. A concurrent team searching for key 26 will see it in its new position at the end of the chunk, without considering the empty entry at all. A team searching for keys smaller than 26 will compute *false* values for the last three entries in the chunk (all of which are greater than the searched-for key), thus ignoring the unsorted portion of the chunk. A team searching for keys smaller than or equal to 21 are guaranteed to reach chunk E before chunk F as long as chunk E has not yet been made a *zombie*, even if keys 17 and 21 have already been copied into F. In fact, this remains true until the pointers to chunk E have been redirected.

Once the merge operation is completed, the team calls *updateDownPtrs* to redirect down-pointers to the *zombie*. The number of chunks in the level is incremented and decremented accordingly.

Deleting From Last Chunk in Level 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 threads in 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.

4.3 Some Words on Correctness

In this section we briefly describe some of the major invariants used by our algorithm. Our main concern is that a traversal will always reach the *enclosing* chunk of the key it is searching for (k) by taking only down and right (lateral) steps. Thus we must ensure when taking a step that we never read a chunk to the right of k's *enclosing* chunk. The following properties, along with the fact mentioned in Section 4.2 that traversals give precedence to higher tIds during `_ballot` operations, aid in ensuring this occurs.

The Max Field Always Decreases One important promise is that the max field of a chunk can only decrease from the moment it is allocated. The max field is ∞ upon allocation, and can be changed in only three places in the algorithm:

- During allocation: Chunks are allocated only during split operations, during which the new chunk receives the max field of the chunk being split. This value is obviously smaller or equal to ∞.
- When the chunk is split: The chunk being split (ch_s) has the top half of its KV pairs moved to the newly allocated chunk. The max field of ch_s is updated accordingly to the largest remaining key, which must be smaller than the max fields former value.
- When the maximum key in a chunk is deleted: The max field will be updated to hold the next-highest value in the chunk.

Insertion can never cause a change in the max field, as a key is only inserted into an *enclosing* chunk, which by definition holds a max field higher than the inserted key. This property is important in ensuring that teams taking lateral steps do not miss the enclosing chunk of a key, as described next.

Lateral Ordering Between Chunks 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 in the same level. This continues to be true even if the key is later deleted from ch. This stems from the previous statement: a key is only inserted into an *enclosing* chunk, and the max field of a chunk only decreases. Thus if a key, k has been placed in ch the *enclosing* chunk of any key larger than k can only be ch itself or a chunk to its right.

Furthermore, this means that a partial order exists between non-*zombie* chunks in any level: a non-*zombie* chunk, ch_{nz}'s NEXT pointer will always point to a chunk containing only keys greater than or equal to the minimum key in ch_{nz}. If ch_{nz} is not currently being split or merged then the minimum key in the next chunk must be greater than the maximum in ch_{nz}. Only during split and merge operations is it possible that ch_{nz} shares some keys with the next chunk.

This remains true even if a chunk becomes a *zombie*. We note that a chunk becomes a *zombie* only as a result of a merge operation, and that the contents of a chunk are

never changed after it becomes a *zombie*. Thus the *zombie*, ch_z, continues to point to the chunk that received its values during the merge that marked it as a *zombie*. The chunk pointed to by the zombie must have had a higher max value at the time ch_z became a zombie. Additionally, any chunks preceding ch_z in the level at that time must have had a lower max value than ch_z. As the max value of a chunk can only decrease, any key greater or equal to the keys ch_z contained when it became a *zombie* can only be inserted into chunks reachable through the NEXT pointer of ch_z. Thus the *enclosing* chunks of all keys that once resided in ch_z, and those of all greater keys, are reachable by taking lateral steps from ch_z.

Order Between Down Pointers Keys are inserted into the structure bottom-up. When k_{in} is inserted into a chunk in level $i+1$, its entry is set to point to the chunk in level i into which k_{in} was inserted. k_{in}'s *enclosing* chunk in level i is either that chunk or a chunk reachable from it through lateral steps. A key can only be moved to a different chunk as a result of a split or a merge operation. In both cases the key will either remain in its original chunk or be moved to a chunk laterally reachable from the original. As the bottom level chunk containing k_{in} is locked k_{in} cannot have been concurrently deleted by another team at the point when k_{in} is inserted into level $i+1$. Thus, after insertion k_{in} and all keys greater than k_{in} are reachable in level i from the chunk reached by taking a down-step through a pointer ssociated with k_{in} in level $i+1$.

The *Delete* operation is performed from the top down with a lock on the containing bottom level chunk. If a key k_d is deleted from level i it follows that it does not exist in level $i+1$, either because it was never raised to that level or because the deleting team already removed it. The lock on the bottom level *enclosing* chunk of k_d ensures that no concurrent *Insert* could have added k_d to levels $i+1$ or higher until the deleting team has concluded its execution. A *Delete* operation does not change the value of entries associated with keys other than k_d. These points, along with the fact that a merge operation can only move keys to a chunk reachable from their original containing chunk, mean that *Delete* operation cannot cause down pointers to point to chunks from which the associated keys cannot be reached laterally.

A call to `updateDownPointers` searches for the *enclosing* chunk of a moved key (k_m), and sets the pointer in level $i+1$ to point to the chunk it discovered. This chunk must either still *enclose* k_m, or k_m's *enclosing* chunk must still be reachable from it (if a split/merge occurred). `updateDownPointers` takes a lock on the chunk in level $i+1$ before updating pointers. This and the top-down order of termDelete operations mean that any key in level $i+1$ whose pointer is updated by *updateDownPointers* must also exist in level i at that time. The key's *enclosing* chunk must, if so, be reachable by taking lateral steps from the chunk pointed to from level $i+1$.

All these taken together help to ensure that a down-pointer associated with k in level $i+1$ always points to a chunk that either *encloses* k or from which k's *enclosing* chunk in level i is reachable by taking lateral steps. Down-steps, if so, can safely be

taken as part of a traversal.

Note that this remains true even in the edge case mentioned in Subsection 4.2.1. In the edge case, a team took a down step associated with a key k_{ds} in level $i+1$, then found that the chunk (ch_i) reached in level i contained only keys greater than k_{ds} (and the key searched for in the traversal, which must be greater than or equal to k_{ds}). This was caused by a concurrent *Delete* operation which removed k_{ds} from both levels between the time the team decided to take the down step in level $i+1$ and the time it read ch_i. At the time k_{ds} was read in level $i+1$ it must still have existed in level i, and have been reachable from ch_i. k_{ds}'s enclosing chunk must still be reachable from ch_i. As ch_i contains only keys greater than k_{ds}, ch_i itself must be k_{ds}'s *enclosing* chunk. Thus, the traversing team succeeded in reaching k_{ds}'s enclosing chunk in level i, albeit without enough data to continue traversing, causing it to restart.

Chapter 5

Measurements/Results

We evaluated GFSL compared to the skiplist algorithm ported to the GPU by Misra and Chaudhuri [MC12b]. The code for their implementation is available online [MC12a]. In the remainder of this section we refer to their implementation as "M&C".

In this work 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 relatively universal to all GPUs, while we optimized the last two to fit our current setup. The values of p_{key} and p_{chunk} are also universal, however, a single best option presented itself in all configurations checked, and so we show results only for those values.

We present benchmarks for GFSL using teams of 32 threads. Chunks are of size 256B with 32 8B key-value pairs, a size which can be read in two transactions. We set a limit on the 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 and with 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".

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. Under this configuration M&C supports two active blocks per SM, with an occupancy of 41.6% out of a theoretical 50%, and 42 registers allocated per threads. We evaluated 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 benchmark a few configurations of M&C showed up to 24% better performance in the 10K range

than those shown in this work. However, *Contains*-only workloads under M&C in small key ranges showed highly unstable performance and very large confidence intervals (up to 50%) . Thus, we chose the configuration that yielded the best results on average.

5.1 Experimental Setup

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.

We tested 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, the *Contains* benchmark runs 10M operations. The number of operations in the *Insert* and *Delete* benchmarks is equal to the key range, i.e. for a 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 (if the operation is not an insert this field is empty). 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.

5.2 Static Configurations

In this section we dive down into the background behind some of our choices for static configurations, e.g the number of warps launched per block, the chunk size in GFSL, and the values of p_{key} and p_{chunk}.

Warps Per Block

We tested GFSL with a varying number of warps launched per block. Table 5.1 shows the results of running a workload with 80% *Contains* operations on a 1M key range as an example of the effects of these different block sizes on throughput and SM utilization. The throughput presented in the table showcases the tradeoff between the amount of concurrency (number of threads launched) and the available resources. We see that the best throughput was achieved for 16 warps launched per block, despite not having the best occupancy or the largest amount of registers per thread.

The *Occupancy* of an SM is the ratio of active warps in an SM to the maximum number of active warps supported by the SM. Theoretical occupancy is the upper limit for the number of active warps an SM can support given the demands of the kernel and the launch configurations. An SM may not be able to achieve the theoretical occupancy if there are many warps that are stalled at the same time (i.e. because of memory transactions that have not completed), leaving no warps eligible for execution by the scheduler. High occupancy may not be an indication of high throughput if other bottlenecks exist, most commonly memory bandwidth.

We see that the achieved occupancy when launching 32 warps per block is very high at around 95%. However, the number of registers allocated per thread is much lower than required. This is evident from two lines in the table: Firstly, we see that when more resources per thread are available, e.g. when fewer threads are launched, the compiler allocates far more registers per thread than 32. Secondly, we see that 53% of all memory bandwidth in this configuration is taken up with access to spillover memory, described in Section 2.2.

On the other hand, allocating enough resources that there is no spillover also does not give maximum throughput, as seen when launching 8 warps per block. In this case there are 79 registers allocated per thread, and there is no spillover to global memory. The low occupancy is the bottleneck in this configuration, as the SM does not have enough active warps that are ready to run to hide the latency caused by warps stalled at any given moment. A major reason for stalled warps is in-progress memory transactions to and from global memory.

In summary, each SM has a finite number of resources, which it distributes equally amongst all threads in all active blocks. Launching more threads naturally means that there are fewer resources per thread. If a kernel requires more resources per thread than available there occurs "spillover" of local variables which are then stored in global memory, causing more global accesses. On the other hand, reducing concurrency too much in order to gain more local resources entails fewer possible concurrently active teams, and reduced benefits from the SM's latency-hiding capabilities. We see that the best balance is reached between concurrency and resource allocation when launching 16 warps per block.

Table 5.2 shows the effects of changing the number of warps per block on M&C. We

Table 5.1: Effects on GFSL of limiting warps launched per block

Warps per Block	8	**16**[1]	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
Local Memory Spillover	0%	**10%**	~43%	~53%
Throughput (MOPS)[2]	58.9	**65.7**	62.5	52.9

[1] The configuration presented in this chapter [2] Throughput for operation mixture [10,10,80], range 1M

Table 5.2: Effects on M&C of limiting warps launched per block

Warps per Block	8	**16**[1]	24	32
Occupancy/Theoretical	52.9%/62.5%	**41.6%/50%**	59%/75%	79.4%/100%
Registers	42	**42**	40	32
Active Blocks	5	**2**	2	2
Local Memory Spillover	~25%	**~23%**	~23%	~24%
Throughput (MOPS)[2]	20.7	**21.3**	20.6	20.2

[1] The configuration presented in this chapter [2] Throughput for operation mixture [10,10,80], range 1M

see that the throughput varies very little, regardless of the number of warps launched or the amount of resources available to each thread. Moreover we see that M&C suffer from spillover even when using the maximum registers deemed sufficient by the compiler. This is most likely because they use thread-local arrays to hold the traversal path. Local arrays are often relegated to spillover memory in CUDA. Further profiling shows that M&C suffers, as expected, from high divergence and inefficient memory alignment and access patterns. Indeed, between 86% and 91% of the latency in M&C's executions is caused by memory dependencies. This indicates that M&C, unlike GFSL, are bound by inefficient memory accesses to the point where they cannot properly utilize available resources on the SM.

Chunk Size

Figure 5.1 illustrates the effects on throughput of executing GFSL with teams/chunks with 32 or 16 threads/entries (GFSL-32 and GFSL-16, respectively). The benchmark presented consists of a workload with 80% *Contains* operations on a range of 1M keys. M&C's results are included for comparison purposes, and are discussed in Subsection 5.3. GFSL-32 and GFSL-16 show similar performance in small ranges, with GFSL-32 outperforming GFSL-16 in the higher ranges by up to 28%. In this work we only allow a single team to run in each warp, regardless the team size.

GFSL-16 uses chunks of size 128B, the maximum size that can be read from global memory in a single transaction. GFSL-32, on the other hand, uses chunks of size 256B which require two transactions. Additionally, GFSL-16 contains 25% more levels on average than GFSL-32. As traversal length is directly tied to structure height, traversals in GFSL-16 theoretically require around 66% fewer memory reads than GFSL-32. Thus

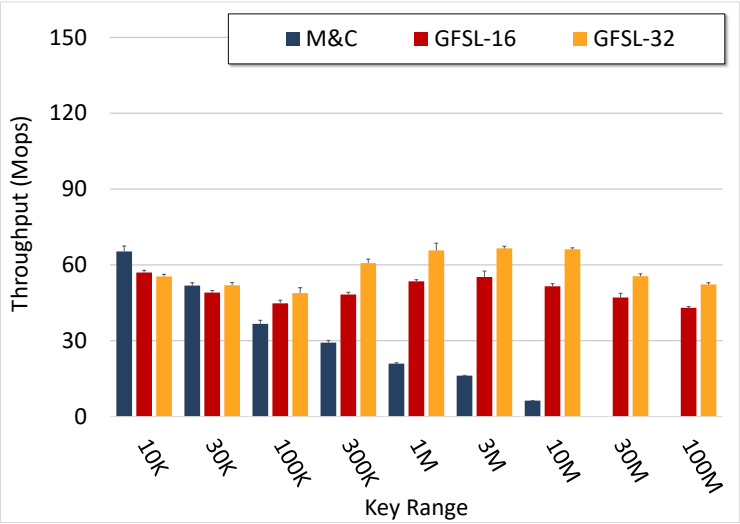


Figure 5.1: Throughput comparison of GFSL using chunks and teams of size 16 (GFSL-16), and of size 32 (GFSL-32), and M&C. The benchmark presented is [i,d,c]=[10,10,80] on a 1M key range

we would expect GFSL-16 to outperform GFSL-32. However, the results presented in Figure 5.1 show that this is not the case. Profiling shows that GFSL-16 uses around half the memory bandwith as GFSL-32. We are unsure of the root cause of the disparity. We believe that GFSL-16 would probably outperform GFSL-32 with proper support for executing two teams within the same warp. However, synchronization between threads in the same warp is a delicate task, as mentioned in Section 2.2. We found that the complexity of the code needed in order to ensure teams within a warp could not deadlock each other caused a significant degradation in performance. Thus this is left for future work.

p_{key} and p_{chunk}

The values of p_{key} and p_{chunk} influence both the number of layers traversed and the number of keys/nodes in each level. We examined the effects of various p_{key} values for M&C between 0.2 and 0.8, and found that in all operation mixtures tested the best results were received for $p_{key} = 0.5$.

Likewise, we found that using $p_{chunk} \approx 1$ in GFSL gave the best results in all operation mixtures tested. This effectively results in $p_{key} \approx 0.05$ for GFSL, as there are 20 entries per chunk on average. In this case the average number of chunks read in a traversal is between $structure - height + 1$ and $structure - height + 2$, meaning that there are between one and two lateral steps taken on average in a traversal. Lowering p_{chunk} causes more lateral steps to be taken, while not having a significant impact on structure height. Thus the overall average traversal is lengthened, causing more global

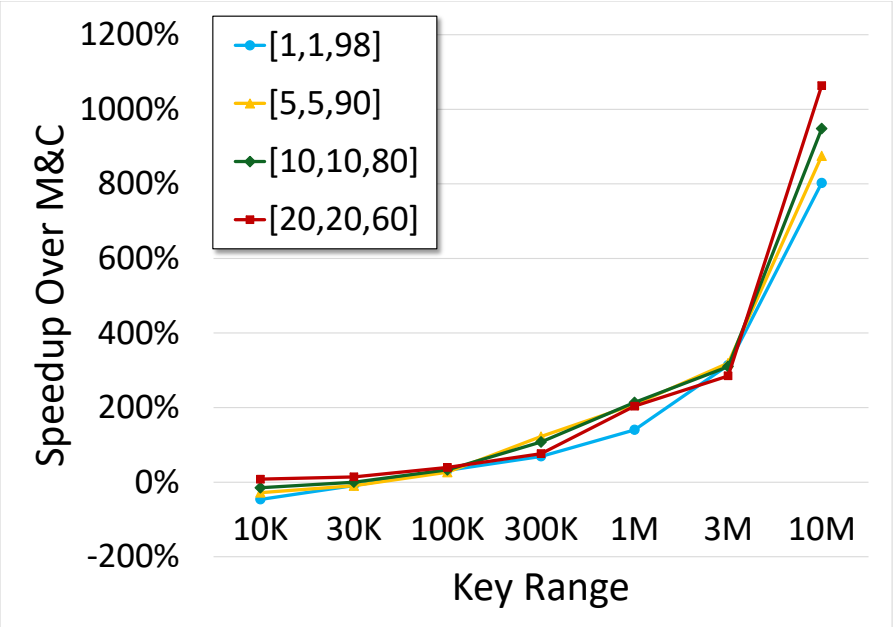


Figure 5.2: Ratio between GFSL and M&C as a function of the key range.

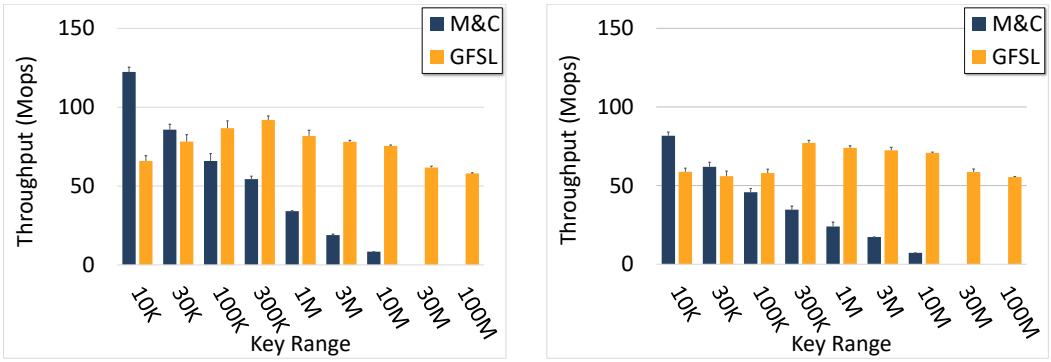
memory accesses.

5.3 Performance Results

Figure 5.2 shows the speedup of GFSL over M&C. GFSL is slower than M&C by up to 46% in the 10K range, up to 10% in the 30K range, then outperforms them by 27% to 1064% in the higher ranges. In Figure 5.3 we present the actual average throughput results and confidence intervals of the various benchmarks. The figure shows that GFSL's performance 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.

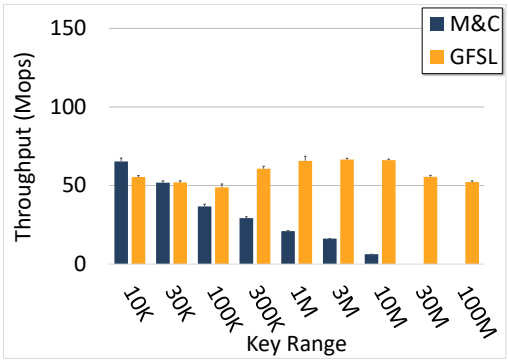
The main advantage of GFSL is the usage of coalesced reads, which optimizes accesses to the global memory. In the smaller range (10K), the entire structure fits into the L2 cache in both implementations, which significantly reduces the benefits of the coalesced reads as L2 access is much faster than 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 69%-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%-46% when the

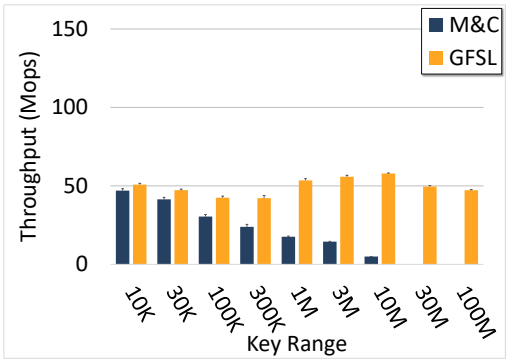


(a) [i,d,c]=[1,1,98]

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



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

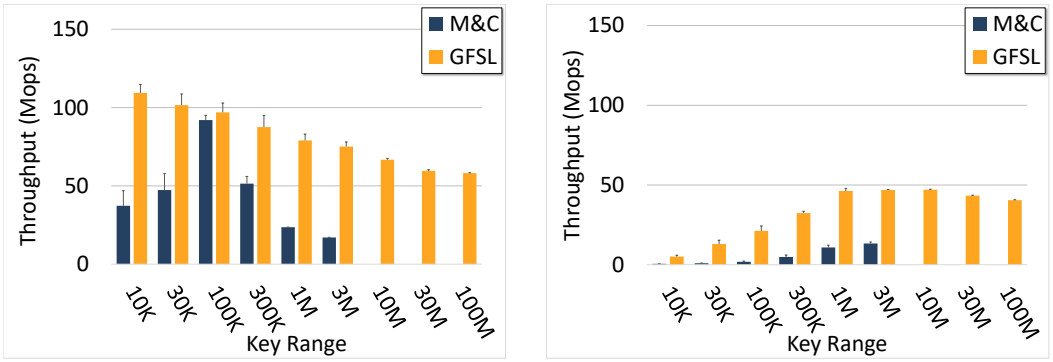


(d) [i,d,c]=[20,20,60]

Figure 5.3: Throughput, in millions of operations per second, as a function of key range.

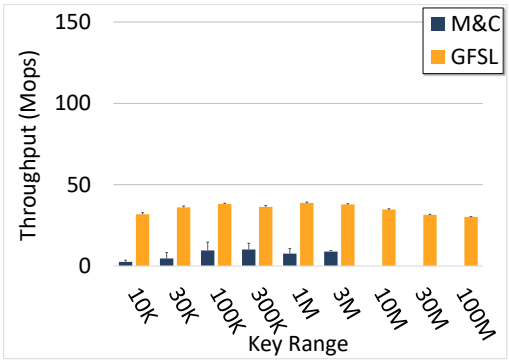
percentage of *Contains* operations is high (Figures 5.3a-5.3c), and slower by 8% when the percentage of *Inserts* and *Deletes* grows (Figure 5.3d). The impact of the distribution is less than the impact of the key range, as GFSL's performance is closer to M&C's in the 30K range then quickly outperforms 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 in the [20,20,60] benchmark). 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



(a) 100% *Contains*

(b) 100% *Inserts*



(c) 100% *Deletes*

Figure 5.4: Throughput, in millions of operations per second, as a function of key range. Each graph shows the throughput of a single operation type.

in Figure 5.4a, which shows the results of the *Contains* test. In this case there are no updates, thus no contention and no dips in GFSL's 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 Figure 5.4. 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 (Figure 5.4a). 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. Figure 5.4b and Figure 5.4c 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*.

Chapter 6

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. [HKOO11] 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. [ZWY+15] 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 [AVS+11, Bor14, KBGB15, BZG+16, GLHL11]. Bordawekar [Bor14] proposed multi-level bounded linear probing, improving locality by using multiple levels of hash tables that reduce the number of lookups. Alcantara et al. [AVS+11] 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, [KBGB15], uses Collaborative Lanes, a method enabling threads in a warp to take on new tasks and so battle warp under-utilization.

Misra and Chaudhuri [MC12b] 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. [CCT12] 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 [SF15] and linked-lists [YHGT10] have been developed for the GPU. Some graph-based algorithms have also been sped up using GPU-optimized implementations [HN07, ZH14, MGG15].

Search trees geared towards graphics applications have also been GPU-optimized to good effect [ZHWG08, LWL12, ZGHG08]. 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 [KCS+10], b+-trees [SCK+11, ZHF14, ZR02], and hash tables [Ros07]. Sprenger et al. [SZL16] 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 [BP11] and B+tree [BP12] 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.

Chapter 7

Conclusion

We presented GFSL, a GPU-friendly algorithm for the skiplist data structure. We identified aspects of classic skiplist algorithms that correlate to known performance drags on the GPU, most importantly lack of memory coalescence and high divergence between threads in a warp. We focused on minimizing these performance drags in the design of GFSL by utilizing 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. GFSL was implemented and evaluated on a GeForce GTX 970 Nvidia GPU (Maxwell architecture). Evaluation shows a speedup of up to 11.6x over previous implementations for large key ranges.

While current results show a significant improvement over the classic skiplist design, further optimizations may help to widen the advantage. For example, some extra concurrency can be gained by finding an efficient way to enable multiple teams in a warp to concurrently handle different operations. This functionality would entail some additional divergence, however the additional computational power may very well overshadow its effects. This is challenging in as teams in the same warp may deadlock while trying to take the lock for the same chunk, as explained in Section 2.2.

Additionally, 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, further expanding the toolkit available to GPGPU programmers.

Bibliography

[ALMS15] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proc. 27th ACM Symp. Parallelism Algorithms Archit. - SPAA '15*, pages 123–132, New York, New York, USA, 2015. ACM Press.

[AVS+11] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. *GPU Computing Gems*, 2:39–53, 2011.

[BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proc. 28th ACM Symp. Parallelism Algorithms Archit. - SPAA '16*, pages 349–359, New York, New York, USA, 2016. ACM Press.

[BNP12] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[Bor14] Rajesh Bordawekar. Evaluation of parallel hashing techniques. *GTC*, 2014.

[BP11] Anastasia Braginsky and Erez Petrank. Locality-Conscious Lock-Free Linked Lists. In *International Conference on Distributed Computing and Networking*, pages 107–118. Springer, 2011.

[BP12] Anastasia Braginsky and Erez Petrank. A Lock-Free B+ Tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67. ACM, 2012.

[Bro15] Trevor Alexander Brown. Reclaiming Memory for Lock-Free Data Structures. In *Proc. 2015 ACM Symp. Princ. Distrib. Comput. - Pod. '15*, pages 261–270, New York, New York, USA, 2015. ACM Press.

[BS10] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on*

General-Purpose Computation on Graphics Processing Units, pages 94–103. ACM, 2010.

[BZG+16] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *USENIX Annual Technical Conference*, pages 281–294, 2016.

[Car13] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[CCT12] Daniel Cederman, Bapi Chatterjee, and Philippas Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. In *European Conference on Parallel Processing*, pages 883–894. Springer, 2012.

[CP15a] Nachshon Cohen and Erez Petrank. Automatic Memory Reclamation for Lock-Free Data Structures. In *Proc. 2015 ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl. - OOPSLA 2015*, volume 50, pages 260–279, New York, New York, USA, 2015. ACM Press.

[CP15b] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proc. 27th ACM Symp. Parallelism Algorithms Archit. - SPAA '15*, pages 254–263, New York, New York, USA, 2015. ACM Press.

[DHK16] Dave Dice, Maurice Herlihy, and Alex Kogan. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. In *Proc. 2016 ACM SIGPLAN Int. Symp. Mem. Manag. - ISMM 2016*, pages 36–45, New York, New York, USA, 2016. ACM Press.

[GLHL11] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. In *ACM Transactions on Graphics (TOG)*, volume 30, page 161. ACM, 2011.

[HKOO11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 46, pages 267–276. ACM, 2011.

[HLLS06] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Provably Correct Scalable Concurrent Skip List. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[HN07] Pawan Harish and PJ Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[KBGB15] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 63–74. IEEE, 2015.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep 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*, pages 339–350. ACM, 2010.

[LWL12] Lijuan Luo, Martin DF Wong, and Lance Leong. Parallel Implementation of R-Trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*, pages 353–358. IEEE, 2012.

[MC12a] Prabhakar Misra and Mainak Chaudhuri. http://www.cse.iitk.ac.in/users/mainakc/lockfree.html, 2012.

[MC12b] Prabhakar Misra and Mainak Chaudhuri. Performance Evaluation of Concurrent Lock-Free Data Structures on GPUs. In *18th IEEE International Conference on Parallel and Distributed Systems*, pages 53–60. IEEE, 2012.

[MGG15] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-Performance and Scalable GPU Graph Traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.

[Nvi15a] Nvidia. CUDA C Best Practice Guide v7.5, September 2015, NVIDIA Developer Zone: website, 2015.

[Nvi15b] Nvidia. CUDA C Programming Guide v7.5, september 2015. NVIDIA Developer Zone: website, 2015.

[Ope15] OpenCL. OpenCL 2.1 Reference Pages, The Khronos Group Inc.: website, 2015.

[Pug90a] William Pugh. Concurrent Maintenance of Skip Lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Science, Department of Computer Science, University of Maryland, 1990.

[Pug90b] William Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[Roc14] RocksDB. A Persistent Key-Value Store for Fast Storage Environments. `http://rocksdb.org/`, 2014.

[Ros07] Kenneth A Ross. Efficient hash probes on modern processors. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1297–1301. IEEE, 2007.

[SCK+11] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[SF15] Thomas RW Scogland and Wu-chun Feng. Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 63–74. ACM, 2015.

[SL00] Nir Shavit and Itay Lotan. Skiplist-Based Concurrent Priority Queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[SO11] Jeff A Stuart and John D Owens. Efficient Synchronization Primitives for GPUs. *arXiv preprint arXiv:1110.4623*, 2011.

[SZL16] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. Cache-sensitive skip list: Efficient range queries on modern cpus. In *International Workshop on In-Memory Data Management and Analytics*, pages 1–17. Springer, 2016.

[WYS+15] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep Image: Scaling up Image Recognition. *arXiv preprint arXiv:1501.02876*, 7(8), 2015.

[YHGT10] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

[ZGHG08] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Highly Parallel Surface Reconstruction. *Microsoft Research Asia*, 2008.

[ZH14] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[ZHF14] Steffen Zeuch, Frank Huber, and Johann-christoph Freytag. Adapting tree structures for processing with simd instructions. 2014.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-Time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

[ZWY+15] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

ביצועים כאשר טווח המפתחות המוכנס לרשימה גדול אך מרע את הביצועים עבור טווח מפתחות קטן. כאשר טווח המפתחות הוא בגודל 10 מיליון, GFSL מציע שיפור בביצועים של פי 6.8-11.6 כתלות בפעולות המבוצעות על הרשימה. לעומת זאת, במקרה הגרוע ביותר של GFSL בטווח מפתחות של עשרת אלפים כאשר מבצעים אחוז גדול של חיפושים ביחס לפעולות עדכון ראינו הרעת ביצועים של פי 0.46. בטווח של 30 אלף מפתחות, GFSL מראה ביצועים דומים או משופרים ביחס ל M&C, ובכל טווח מפתחות יותר גדול מ 30 אלף GFSL מראה שיפור ביחס ל־M&C.

היתרון המרכזי של GFSL הוא בקריאות המאוחדות של המידע, המאפשרות גישה יעילה לזכרון הראשי האיטי לקריאה של המעבד הגרפי. לעומת זאת, עבור טווח מפתחות קטן מבנה הרשימה קטן ולכן נכנס כולו בזיכרון המטמון של המעבד הגרפי (L2 Cache). במקרה זה, קריאות המידע (Memory Transactions) לרוב אינן מגיעות לזכרון הראשי, ולכן אנו פחות מרוויחים מאיחוד הקריאות שהמבנה שלנו מאפשר. השיפור המשמעותי שעולה במחקרנו מופיע כאשר הרשימה גדולה מכדי להיכנס כולה לזיכרון המטמון, ולכן מצריכה קריאות וכתיבות רבות לזיכרון הראשי.

תוצאה מעניינת היא העובדה כי בעוד הביצועים של M&C יורדים בצורה דרסטית ככל שטווח המפתחות גדל, GFSL מראה ירידה מתונה בביצועים בלבד. בטווחים הכי גדולים שנבדקו במחקר זה (30 מיליון ו100 מיליון מפתחות) ראינו כי המימוש של M&C אינו מצליח להתמודד עם העומס וקורס בשל מגבלות זכרון, בעוד ש GFSL מצליחה להתמודד וממשיכה להראות ירידת ביצועים מתונה.

רשימת דילוגים מורכבת מאוסף של רשימות מקושרות ממויינות, המסודרות כך שכל רשימה מהווה "רמה" במבנה. הרמה התחתונה של הרשימה מכילה את כל המפתחות המוכלים במבנה ממויינים בסדר עולה. כל רמה, פרט לתחתונה, מכילה תת קבוצה של המפתחות הנמצאים ברשימה ברמה מתחתיה. חיפוש מפתח במבנה מתבצע על ידי חיפושו בכל רמה החל מהרמה העליונה, כאשר כל צעד ברשימה ברמה מסויימת "מדלג" על מספר מפתחות ברמה התחתונה ובכך מקצר את פעולת החיפוש. לדוגמה, מפתח הנמצא ברמה i כלשהי במבנה יופיע גם ברמה $i+1$ בהסתברות p_{key} הנקבעת על ידי המשתמש, כאשר נהוג להשתמש בערך p_{key} המקיים: $1/4 <= p_{key} <= 1/2$.

לרשימת דילוגים בצורתה הקלאסית יש חסרונות רבים בסביבה של המעבד הגרפי. חיסרון מרכזי שבו נתקלים בהסבה ישירה של רשימת דילוגים לריצה על מעבד גרפי הוא שרשימת דילוגים מתאפיינת בכך שהמידע בה מפוזר ברכבי הזכרון, והגישה למידע אינה סדירה. בנוסף, רשימה זו דורשת סנכרון ברמת החוט הבודד, פעולה מאוד איטית במעבד הגרפי. חסרונות אלו תורמים לכך שהסבה ישירה של רשימת דילוגים אינה יעילה מבחינת ביצועים.

אנו מציעים את GFSL, רשימת דילוגים המותאמת לריצה על המעבד הגרפי ומבוססת נעילות בגרעיניות נמוכה (Fine-Grained Locking). GFSL מורכבת מאוסף של רשימות מקושרות ממויינות, כאשר כל צומת ברשימה הוא מערך חד ממדי בשם "אסופה" (chunk). כל אסופה מכילה מספר מפתחות עוקבים הממויינים בסדר עולה. בדומה לרשימת דילוגים סטנדרטית, כל המפתחות המוכלים במבנה קיימים ברמה התחתונה, וכל רמה אחרת מכילה תת קבוצה של המפתחות ברמה מתחתיה. בנוסף לכל מפתח באסופה מוצמד ערך. באסופות הנמצאות ברמות העליונות של הרשימה, ערך זה הוא מצביע לאסופה ברמה מתחת, וברמה התחתונה ביותר ערך זה הוא המידע שהוכנס לרשימה יחד עם המפתח ומיוצג על ידו. בכל אסופה ישנם שני שדות נוספים, מלבד רשימת הצמדים של מפתחות וערכים: שדה של מנעול ושדה שמצביע לאסופה הבאה באותה רמה.

במילים אחרות, כל צומת ב־GFSL מכיל מספר מפתחות, בשונה מהמימוש הסטנדרטי של רשימת דילוגים בו כל צומת מכיל מפתח אחד בלבד. בנוסף, ב־GFSL ההחלטה אם להוסיף מפתח לרמה $i+1$ מתבצעת רק כאשר נוצרת אסופה חדשה ברמה i (בהסתברות p_{chunk}). לעומת זאת, ברשימה סטנדרטית מחליטים האם להוסיף מפתח לרמה $i+1$ בכל פעם שמתווסף מפתח ברמה i (בהסתברות p_{key}). התוצאה הינה שב־GFSL מעלים פחות מפתחות לרמה $i+1$ ולכן GFSL מכילה פחות צמתים ופחות רמות מאשר קיימים ברשימת דילוגים סטנדרטית בעלת הסתברות $1/4 <= p_{key} <= 1/2$ המכילה אותו מספר מפתחות.

במחקר זה אנו מחלקים את החוטים לקבוצות לוגיות הנקראות צוותים (Teams). מספר החוטים בצוות הינו קטן או שווה למספר החוטים בwarp של המעבד הגרפי. כל החוטים באותו צוות מבצעים גישה מאוחדת לאסופות ומשתפים פעולה בביצוע פעולה בודדת על רשימת הדילוגים. בעזרת גישה מאוחדת זו אנו מצמצמים את כמות הפעולות שניתן לבצע במקביל על הרשימה, אך לעומת זאת אנו משיגים גישה סדרתית לזכרון ופחות התבדרות בבקרת הזרימה של התוכנית. צורת מעבר זאת על רשימת הדילוגים בה החוטים בקבוצה משתפים פעולה מתאימה לחוזקות של המעבד הגרפי בכך שניתן לעבד כמות מידע גדולה בכל צעד בחישוב.

בכדי לבדוק את GFSL אנו משווים אותה למימוש רשימת דילוגים אשר נכתב ע"י Misra and Chaudhuri [MC12b], אשר הוכח כמשפר ביצועים בהשוואה למימוש מקביל על המעבד המרכזי. להלן נכנה את המימוש שלהם כM&C. התוצאות במחקר הנוכחי מראות כי המימוש והאופטימיזציות שלנו משפרות

תקציר

בשנים האחרונות מעבדים גרפיים הפכו לכלי פופולרי להאצת תוכניות מחשב. היום מעבדים גרפיים מספקים מאות ליבות חישוב חסכוניות באנרגיה במחיר נמוך, ומספר ליבות החישוב עולה בכל דור של מוצרים אלה. בנוסף, הוצאתן של סביבות פיתוח מותאמות לתכנות מקבילי על גבי מעבדים גרפיות כמו CUDA ו־ OpenCL תרמה רבות לאפשרות לשימוש במעבדים גרפיים לצורכי עיבוד וחישוב כלליים (GPGPU), ללא צורך ברקע בתכנות גרפי.

העניין בפיתוח תוכנות GPGPU זינק בשנים האחרונות, וכיום המעבד הגרפי משמש כמאיץ לאפליקציות במגוון רחב של תחומים, החל מלמידה עמוקה (Deep Learning) וכלה בפעולות על מסדי נתונים. עם זאת, קיים עדיין אתגר משמעותי בתכנון ובמימוש של אלגוריתמים כלליים ויעילים עבור מעבדים גרפיים. אתגר זה נובע מהאופן שבו מעבדים גרפיים פועלים, השונה במהותו מאופן פעולת ההמעבד המרכזי של המחשב (CPU). מעבדים גרפיים מאוד יעילים עבור גישה "רגולרית" (סדרתית ומסודרת) למידע שעליו מתבצע החישוב, בעיקר כאשר המידע מיוצג כאוסף של וקטורים ו/או מטריצות. בנוסף, מעבדים גרפיים מריצים חוטים במקבצים הנקראים warps, ולא כחוטים בודדים כפי שנהוג במעבד המרכזי. דבר זה נותן יתרון לאחידות של בקרת הזרימה בתוכניות הרצות בסביבת המעבד הגרפי. לעומת זאת, גישה לא סדרתית לזיכרון או התבדרות בקרת הזרימה בתוכניות יכולות להוביל לפגיעה משמעותית בביצועים. צורות התנהגות אלו מאוד נפוצות בקרב אפליקציות ואלגוריתמים אשר משתמשים במבני נתונים מבוססי מצביעים (pointer-based data structures) אשר תומכים בעדכונים שוטפים. שימוש במבנים מבוססי מצביעים כאלו מאוד נפוץ באלגוריתמים כלליים.

קיימים אלגוריתמים ומימושים רבים של מבני נתונים מקביליים מבוססי מצביעים המיועדים לשימוש במעבד המרכזי. עם זאת, ניסיונות לבצע הסבה ישירה של מבני נתונים אלו לריצה על מעבדים גרפיים הראו כי ברוב המקרים יש צורך בעוד אופטימיזציות בכדי להתאים אותם לסביבת המעבד הגרפי ובכך לשפר את ביצועיהם.

אנו מאמינים כי המעבד הגרפי יוכל בעתיד לתת שירותים מורכבים למעבד המרכזי, כגון, הידור במהלך הריצה (JIT compilation) ושחרור זכרון אוטומטי (Garbage Collection). על מנת שמפתחי תוכנה יוכלו להשתמש במעבד הגרפי לצרכים כאלו, יש תחילה לפתח עבור סביבה זו את הכלים המהווים את אבני הבניין הבסיסיות של תכנות. כלים אלו כוללים מבני נתונים כגון רשימת דילוגים (skiplist).

רשימת דילוגים הינה מבנה נתונים פופולרי במימוש אלגוריתמים מקביליים, כיוון שהיא מספקת חלופה הסתברותית עבור עצי חיפוש מאוזנים ללא צורך בפעולות איזון מסובכות, אשר קשות ויקרות לביצוע בסביבה מקבילית. דוגמאות לשימוש נפוץ ברשימות אלו הן אלגוריתמים לאחסון מפתח־ערך (Key-Value Stores) ובסיס למבני נתונים מקביליים אחרים כגון תור־קדמויות (Priority Queue).

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחברת ושותפיה למחקר בכנסים ובכתבי־עת במהלך תקופת תואר המגיסטר של המחברת, אשר גרסאותיהם העדכניות ביותר הינן:

Nurit Moscovici, Nachshon Cohen, and Erez Petrank. A gpu-friendly skiplist algorithm. In *International Conference on Parallel Architecture and Compilation Techniques (PACT), 2017*, 2017. (in press).

Nurit Moscovici, Nachshon Cohen, and Erez Petrank. Poster: A gpu-friendly skiplist algorithm. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 449–450. ACM, 2017.

המאמר נבחר כמועמד לפרס המאמר הטוב ביותר בPACT 2017, אשר עוד לא התקיים בזמן כתיבת חיבור זה.

אני מודה לטכניון על התמיכה הנדיבה במשך השתלמותי

רשימת דילוגים המותאמת לסביבת המעבד הגרפי

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

נורית מושקוביץ

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
תמוז התשע"ז חיפה יולי 2017

רשימת דילוגים המותאמת לסביבת המעבד הגרפי

נורית מושקוביץ