A Skip List Data Structure Library for GPU

Taoming Liu
tl3109@nyu.edu

Jicheng Yu
jy2575@nyu.edu

Zitong Xuan
zx1606@nyu.edu

ABSTRACT

This project presents an implementation of a skip list data structure optimized for Graphics Processing Units (GPUs) using CUDA (Compute Unified Device Architecture). The increasing demand for high-speed data processing and concurrent data structures in parallel computing necessitates efficient and scalable GPU-based solutions. Our work focuses on leveraging the massively parallel architecture of GPUs to enhance the performance of skip lists, a probabilistic alternative to balanced trees, widely used in scenarios requiring fast search, insert, and delete operations. Our group has tried two different implementations for a concurrent skip list, and we finalized on a lock-free implementation for its simplicity and performance. We landed on a maximum of 3328.71 speedup in a search-heavy, skip list operations configuration for a total 1000k operation counts when running on NYU CIMS CUDA3 server.

Keywords: Graphic Processing Units(GPU); Skip List; CUDA Programming; Lock-free Data Strucures

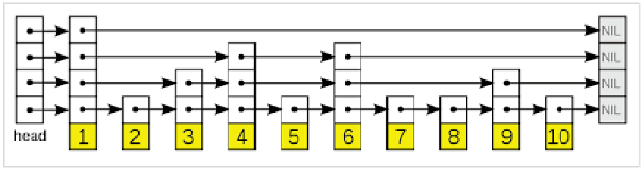

Figure 1: Skip list: A linked list with some randomly-chosen shortcuts

1 INTRODUCTION

Skip list is an efficient and probabilistic alternative to traditional structures like balanced trees. Originally introduced by William Pugh[1] in 1989, skip list offers a compelling blend of simplicity and performance, making it suitable for various applications in computer science and related fields.

Essentially, a skip list is a layered data structure that facilitates fast search, insertion, and deletion operations, akin to those in a sorted linked list, but with enhanced efficiency. The structure consists of multiple layers of linked lists, where each successive layer is a subsequence of the previous one. The bottom layer represents the entire set of elements, while each higher layer provides a progressively streamlined view, skipping over a number of elements to provide a form of "express lane" for traversing the data. Figure 1 shows a basic configuration for a linked-list based skip list.

Skip list is very efficient in operations including search, insertion, and deletion, enabling these to be performed in $O(\log n)$ time on average with high probability. This is due to its probabilistic balancing, where the level of each element is randomly determined, ensuring a balanced structure on average. For example, a search operation starts at the highest layer, moving downwards and across, exploiting these shortcuts to achieve logarithmic time complexity.

Insertion and deletion operations follow a similar logic. These operations entail first locating the position where the change is to occur, following the logarithmic search pattern. After identify the location, the skip list will modify itself accordingly: In insertion, the skip list will add new node at the bottom level, copy itself to upper levels determined probabilistically and update its adjacent nodes accordingly. In deletion, the skip list will remove all nodes appeared at each levels and update the adjacent nodes. The randomness in assigning node levels plays a key role in maintaining the skip list's balanced structure on average. As a result, both insertion and deletion activities can be performed with an expected time complexity of $O(\log n)$.

Skip lists have many usages in various domains such as database indexing, network routing, and memory management. In database systems, skip lists serve as an alternative to B-trees for indexing, offering simplicity in implementation and efficient performance, particularly in scenarios involving concurrent access. Network routing algorithms also leverage the skip list's structure for quick pathfinding and update operations, which are crucial in dynamic networking environments. Furthermore, in memory management, skip lists are employed for efficient allocation and retrieval of memory blocks due to their ability to quickly adjust to changes in the dataset.

The advent of General-Purpose computing on Graphics Processing Units (GPGPU) has opened new avenues for enhancing the performance of data structures like skip lists. Utilizing GPUs for skip list operations exploits the massive parallelism inherent in these devices, significantly accelerating the process. In a GPU-enabled skip list, operations such as search, insertion, and deletion can be distributed across thousands of threads, allowing simultaneous processing of multiple elements. This parallel processing capability is especially beneficial for large datasets, where the traditional sequential approach would be prohibitively slow.

In this project, we utilize CUDA (Compute Unified Device Architecture) to implement a lock-free parallel skip list, taking full advantage of the parallel computing capabilities of NVIDIA GPUs. CUDA provides a comprehensive development environment and tool suite that allows for direct programming of the GPU's virtual instruction set and memory. Our implementation aims to demonstrate significant improvements in performance for common operations like search, insertion, and deletion, particularly in large datasets where traditional CPU-based approaches fall short. By adopting a lock-free design, we further ensure that the data structure minimizes bottlenecks and maximizes efficiency, making it suitable for real-time applications and large-scale data processing tasks.

2 BACKGROUND AND RELATED WORKS

2.1 Atomic Operations

In our project, we extensively utilize atomic operations to maintain the integrity of our skip list, particularly safeguarding it against race conditions. Atomic operations are integral in concurrent programming, where they ensure the atomicity of accesses and updates to shared single-word variables. These operations are crucial in multi-threaded environments, where multiple threads may attempt to read or modify shared data simultaneously.

In particular, we use *atomicCAS()* function, a widely-used atomic operation in CUDA programming that atomically compares the content of a memory location with a given value and, if they match, modifies the content to a new specified value. This operation helps our lock-free skip list to avoid race conditions, ensuring that two threads do not simultaneously modify the same node, which could lead to inconsistent or corrupt data structures.

2.2 Related Works

We implemented a lock-free skip list in this project. While a significant amount of work has been done on lock-free data structures, there is relevantly little work done on lock-free skip list data structure on GPUs. We studied on several existing parallel implementation on skip-list. Fraser devised a CAS-based lock-free skip list by treating each level of a skip list as a linked list[2]. Herlihy and Shavit further provided a detailed implementation of a lock-free skip list in java on CPU based on Fraser's design[3]. In our project, we borrowed some ideas from Herlihy and Shavit's implementation and modified their construction to a GPU-friendly lock-free skip list using CUDA.

3 METHODS

Our implementation of the lock-free skip list is based on the linearizable design outlined in[3]. A Search operation performs as a regular search in sequential skip list. First, we scan for the target in the shortcut list, starting at the $-\infty$ sentinel node. If we find the target, we're done. Otherwise, we reach some value bigger than the target and we know that the target is not in the shortcut list. Then we scan for the target in the original list and output the result. Figure 2 shows a Search operation for node 5. Algorithm 1 presents a pseudocode for our Search operation.

Further, we use atomic operations to ensure correctness of our Insert and Delete operations. To insert a new key 'a' using the Insert(a) function in a skip list, the function first determines the appropriate position for insertion by navigating through the skip list. Then, it randomly decides a level for this key. Subsequently, a new node is created and added to each of the linked lists from level zero up to this level. the Delete(x) function for an existing key 'x' in the skip list first locates the key and then proceeds to eliminate the node from all the linked lists in which it is included. Algorithm 2 and Algorithm 3 shows an implementation for Search and Delete operation using *atomicCAS()* atomic operation.

4 EXPERIMENTAL SETUP

Our experiments were conducted on the skip list implemented as we introduced in Section 3. The skip list was parameterized with a probability $p = 0.5$ and configured to maintain 32 levels.

The computational experiments were structured around two primary operation sets: Set A with a ratio of $4:4:2$ for *Insert* : *Delete* : *Contain* operations and Set B with a ratio of $2:2:6$. Each

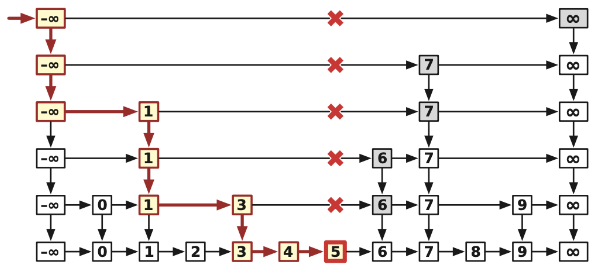

Figure 2: Searching for 5 in a list with shortcuts

Algorithm 1 Search operation in a Skip List

```
 1: function SEARCH($key$)
 2:     $bottom \leftarrow 0$
 3:     $mark \leftarrow$ false
 4:     $prev \leftarrow head$
 5:     $curr \leftarrow$ null
 6:     $next \leftarrow$ null
 7:     for $level \leftarrow MAX\_LEVEL$ downto $bottom$ do
 8:         $curr \leftarrow prev$.GETREFERENCE($level$)
 9:         while true do
10:             $next \leftarrow curr$.GET($level, mark$)
11:             while $mark$ do
12:                 $curr \leftarrow curr$.GETREFERENCE($level$)
13:                 $next \leftarrow curr$.GET($leve, mark$)
14:             end while
15:             if $curr.key < key$ then
16:                 $prev \leftarrow curr$
17:                 $curr \leftarrow next$
18:             else
19:                 break
20:             end if
21:         end while
22:     end for
23:     return $curr.key == key$
24: end function
```

set encompassed a variety of configurations characterized by a combination of five key ranges [100, 1000, 10000, 100000] and five distinct total operation counts [10000, 50000, 100000, 500000, 1000000], resulting in 25 unique scenarios per operation set.

Each configuration was executed using a CUDA kernel with 512 threads per block and 64 operations per thread, a decision made to optimize parallel throughput and maintain consistency across tests. The performance was measured in terms of speedup, calculated by comparing the execution time of the CUDA implementation against its sequential counterpart.

The computational environment for our experiments was provided by NYU's CUDA GPU Computing Servers. Our experiments aim to shed light on the scalability and efficiency of lock-free skip lists in high-throughput, parallel computing environments.

Algorithm 2 Insertion operation in a Skip List

```
function Add($key, numItems$)
    $newNode \leftarrow$ GetNewNode($key, numItems$)
    $top \leftarrow newNode.topLevel$
    $bottom \leftarrow 0$
    $prevs \leftarrow$ array of Node pointers of size MAX_LEVEL+1
    $nexts \leftarrow$ array of Node pointers of size MAX_LEVEL+1
    $t \leftarrow$ false
    while true do
        $found \leftarrow$ find($key, prevs, nexts$)
        if $found$ then
            return false
        else
            for $level \leftarrow bottom$ to $top$ do
                $succ \leftarrow nexts[level]$
                $newNode$.SetRef($level, succ$, false)
            end for
            $prev \leftarrow prevs[bottom]$
            $succ \leftarrow nexts[bottom]$
            $t \leftarrow prev$.CAS($bottom, succ, newNode$, false, false)
            if not $t$ then
                continue
            end if
            for $level \leftarrow bottom + 1$ to $top$ do
                while true do
                    $prev \leftarrow prevs[level]$
                    $next \leftarrow nexts[level]$
                    if $prev$.CAS($level, next, newNode$, false, false) then
                        break
                    end if
                    find(key, prevs, nexts)
                end while
            end for
            return true
        end if
    end while
end function
```

Algorithm 3 Deletion operation in a Skip List

```
function Delete($key$)
    $bottom \leftarrow 0$
    $mark \leftarrow$ an array initialized with false
    $prevs \leftarrow$ array of Node pointers of size MAX_LEVEL+1
    $nexts \leftarrow$ array of Node pointers of size MAX_LEVEL+1
    $next \leftarrow$ null
    while true do
        $found \leftarrow$ find($key, prevs, nexts$)
        if not $found$ then
            return false
        else
            $curNode \leftarrow nexts[bottom]$
            for $level \leftarrow curNode.topLevel$ to $bottom + 1$ do
                $next \leftarrow curNode$.Get($level, mark$)
                while $mark[0]$ = false do
                    $curNode$.CAS($level, next, next$, false, true)
                    $next \leftarrow curNode$.Get($level, mark$)
                end while
            end for
            $next \leftarrow curNode$.Get($bottom, mark$)
            while true do
                $marked \leftarrow curNode$.CAS($bottom, next, next$, false, true)
                $next \leftarrow nexts[bottom]$.Get($bottom, mark$)
                if $marked$ then
                    find(key, prevs, nexts)
                    return true
                else if $mark[0]$ then
                    return false
                end if
            end while
        end if
    end while
end function
```

5 EXPERIMENTAL RESULTS

In our experimental evaluation of the lock-free skip list, we focused on the speedup performance across various configurations of operation sets A and B, with the speedup calculated by comparing the kernel execution times of parallel CUDA implementation with the execution time of the sequential counterpart.

In operation set A, a significant speedup was observed, reaching up to 3357.44 for 1,000,000 operations at the lowest key range of 100. As the total number of operations increased, the speedup also rose, showing the scalable nature of the CUDA-based skip list. This scalability is particularly clear when noting that even at ten thousand operations, a key range of 100,000 still produced a speedup of 2.24, suggesting a baseline improvement over sequential execution.

Set B, tailored for read-heavy workloads with its operation ratio of 2:2:6 for Insert:Delete:Contain, also showed a pronounced speedup. The peak speedup was at 1468.15 for 1,000,000 operations at a key range of 100. The trend of increasing speedup with the growing total number of operations held true for Set B as well. However, for a fixed operation count, we observed an increase in speedup as the key range decreased, highlighting the efficiency of the CUDA architecture in dealing with denser key spaces.

When comparing operation sets A and B with the same configurations, Set A generally showed higher speedup rates. For example, with one million operations at a key range of 1000, Set A realized a speedup of 1780.78, while Set B reached 774.04. This marks a significant performance advantage in favor of Set A. The reasoning behind these results can be partly attributed to the increase in atomic operations as the key range increases. The atomic add and delete operations call for the modification of the data structure through atomicCAS operations. As a result, as the key range expands, more atomicCAS operations are required, leading to a notable drop in speedup due to the increased complexity of control flow within the implementation. However, it's worth noting that a larger proportion of add operations, as in the 4:4:2 ratio of Set A, is associated with an increase in speedup. The increase in add operations likely leads to an higher number of shortcuts within

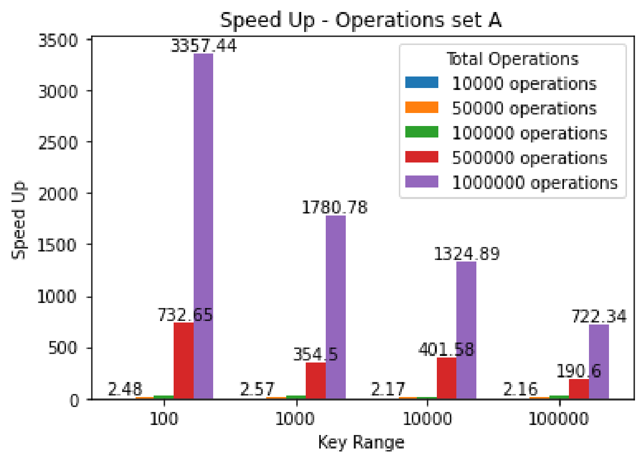

Figure 3: Speedup of skip list on GPU

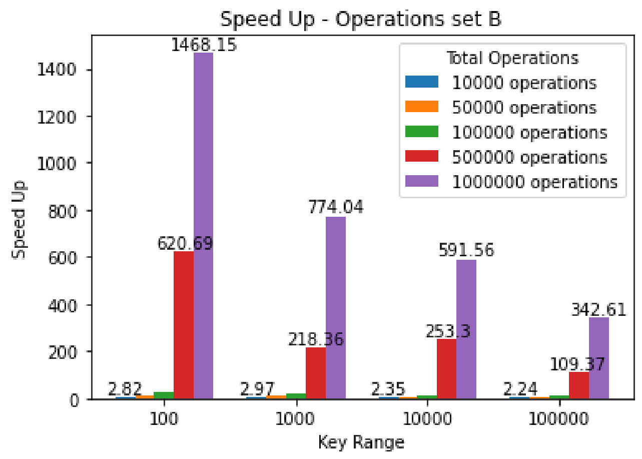

Figure 4: Speedup of skip list on GPU

the skip list, effectively reducing the number of node traversals required and thus making the control flow more direct.

This analysis of the speedup performance across the two operation sets not only highlights the benefits of GPU parallelism in concurrent data structures but also shows the influence of different operation proportions and key ranges on the skip list's efficiency. It confirms the potential benefits of using CUDA for concurrent operations, notably in scenarios dominated by read operations.

6 OTHER IMPLEMENTATION

We also tried to implement a more efficient skip list based on the paper "A GPU-Friendly Skiplist Algorithm"[4]. In this paper, Moscovici, Petrank, and Cohen proposed another way of implementing the skip list on GPU. This implementation is said to have higher memory coalescence and lower execution divergence. It does not use the link-list format, but changes each level of data into an array-based structure called "chunks". Each chunk contains a data array which stores the key-value pairs. And a NEXT field and a lock field, for simplicity we use an array of key-value pairs and set the last two fields to be NEXT and lock. For the data field, if the chunk is not at the bottom level, then the value stores the pointer to the chunk that encloses the key in the lower level. For the first chunk and each level, the first data field does not store a key-value pair but the pointer to the first chunk in the lower level. The pointer to each chunk is an index number. We pre-allocate a pool of chunks and the chunks are stored in an array, so an index to this array could get the corresponding chunk. We use an index variable to track the next usable chunk. When a new chunk is needed to the skip list, we just atomic increment this index variable and return it as the index to the chunk.

This skip list contains a head array that stores the pointer to the first chunk of each level and the number of chunks in each level. During insertion and deletion, the number of chunks may increase and decrease at each level.

The skip list performs each operation by a "team", which consists of several threads. Normally the size of the team is the warp size. Each thread in the warp is related to a field in the chunk and will need to deal with the data in that field during the operations.

For the contains operation, the team will first find the highest level that contains a non-empty chunk. Then starting from the first chunk in each level, each thread in the team will read the data in the chunk and decide whether to move to the next chunk in this level or the chunk in the lower level. The threads communicate with each other by a warp-level function __ballot_sync(). It will search until the bottom level, and check if the key is in the bottom level starting from a chunk reached from the higher level.

The insertion operation starts from a similar search in contains. But besides whether the key is in the skip list, it also returns a "path" to that key. A path is a set of chunks at each level that enclose the key. "Enclose" means the key is less or equal to the max field in this chunk. In this implementation, the path is not an array of chunks but is a local variable that is the pointer of the chunk in each thread. And the thread with warp id = level_index holds the path chunk of this level. The path is obtained from another warp-level function __shfl__sync(). With the path, starting from the bottom chunk, the key is inserted into the chunk. If the chunk is not full, the key could be inserted directly. But if the chunk is full, a split operation is needed and a key could be raised to a higher level. The split operation creates a new chunk and copies the top half of the key-value pairs in the old chunk to the new chunk. It also redirects the pointers of those key-value pairs to the new chunk.

The delete operation is similar to the insert operation. But instead of splitting a chunk, it may cause a merge of two chunks. And it needs to deal with the situation that the key to be deleted is the only element in the last chunk of each level.

We tried to implement the contains and insertion operations. However, due to the complexity of the code and warp-level communications between the threads, we do not have enough time to make the code work properly.

7 CONCLUSION

This project successfully demonstrates the significant potential of GPU-accelerated skip lists, particularly in the context of concurrent data structures. By implementing a lock-free skip list using CUDA, we have not only addressed the challenges associated with concurrent operations in traditional CPU-based data structures but have also harnessed the parallel processing capabilities of GPUs. Our experimental results provide compelling evidence of the efficiency and scalability of our approach. The observed speedups, especially in configurations with high operation counts and lower key ranges, underline the effectiveness of CUDA-based parallelism in managing large-scale data processing tasks.

Furthermore, the comparison of different operation sets in our experiments highlights the adaptability of our implementation to various operational demands. The significant speedup in both search-heavy and balanced operation sets demonstrates the versatility of the GPU-optimized skip list in handling diverse workloads.

However, the project also encountered challenges, particularly in the more advanced implementation based on Moscovici, Petrank, and Cohen's design. While our initial implementation of this concept did not reach full functionality within the project timeline, it nevertheless points to a promising direction for future research and development. Overcoming these challenges could lead to even more efficient GPU-accelerated data structures, leveraging both the raw computational power of GPUs and the more sophisticated aspects of their architecture.

REFERENCES

[1]William Pugh. Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6):668–676, 1990.

[2]K. Fraser. “Practical Lock-Freedom”. PhD dissertation, Kings College, University of Cambridge, September 2003.

[3]M. Herlihy and N. Shavit. “The Art of Multiprocessor Programming”. Morgan Kaufmann Publishers.

[4]N. Moscovici, N. Cohen and E. Petrank, "A GPU-Friendly Skiplist Algorithm," 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, USA, 2017, pp. 246-259, doi: 10.1109/PACT.2017.13.