A Lock-Free Concurrent Linked List Library on GPU

Yibo Liu
University of Victoria
V01036997
liuyibo@uvic.ca

Nasim Pak
University of Victoria
V01010824
zpak@uvic.ca

Jose Guerrero
University of Victoria
V00900773
guerrej@uvic.ca

Abstract

The goal of this project is to build a singly linked list library for GPUs with all operations implemented in CUDA which fully utilizes the parallelism of GPUs. Our implementation of a locked-free linked list algorithm on CUDA is novel, since it was only implemented on multi-core processors in the previous works.

Keywords

Lock-Free Algorithms, Concurrent Linked Lists, GPU Programming, CUDA, Parallelism

ACM Reference Format:
Yibo Liu, Nasim Pak, and Jose Guerrero. 2024. A Lock-Free Concurrent Linked List Library on GPU. In *Proceedings of University of Victoria CSC586B Project (CSC586B Project '24).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The goal of this project is to build a singly linked list library for GPUs with all operations implemented in CUDA which fully utilizes the parallelism of GPUs. In general, a linked list outperforms an array in insertion and deletion operations, the time complexity of a linked list is O(1), while it of an array is O(n). In the scenario with many insertion and deletion operations happening simultaneously, a traditional linked list operates sequentially. To further improve the performance, algorithms for concurrent linked lists are already proposed.

The difficulty for operating insertion and deletion of nodes concurrently in a linked list lies in the conflict of different threads accessing the same address. A straightforward solution is adding lock to the nodes or the threads, making the accesses mutual exclusive[1]. However, locked-based algorithm is ineffective as threads blocking for each other by the locks, especially for GPU which contains thousands of threads [6]. A more delicate solution is lock-free synchronization. For lock-free concurrent linked list algorithms, [7] first proposed a concurrent linked list algorithm, he made use of auxiliary nodes to solve the problem of concurrently deleting two adjacent nodes. Then Harris[5] proposed another algorithm, in which logical marks are used to solve problems in concurrent insertion and deletion. There are further improvements after Harris', for example, [9] added a subsequent traversal of the list which does garbage collection of logically deleted nodes. [6] expanded the lock-free algorithm to more data structures on GPU, including linked list, hash table, skip list, and priority queue.

In this project we implemented the Harris' algorithm for lock-free concurrent linked list with CUDA. A sequencial version linked list is also implemented with C++ as a baseline. Our codebase can be found at [2]. Open-source implementations of Harris' algorithm with PThreads and OpenMP are used as references for this CUDA implementation, they can be found at [3] and [4].

2 Background

A thorough explanation of all possible problems that a concurrent linked list will encounter and the solution by Harris' algorithm is described in the paper [5]. Here we briefly summarize Harris' solution to concurrent maintenance of ordered linked lists that is non-blocking, using a CAS primitive. Insertion of node n after node *p*:

1. *next* ← *p->next*
2. *n->next* ← *next*
3. CAS(address-of(*p->next*), *next*, n)
4. If the CAS operation was not successful, go back to 1.

Deletion of *p->next* is more involved. The naive solution of resetting this pointer with a single CAS runs the risk of losing data when another thread is simultaneously inserting. Instead, two invocations of CAS are needed for a correct algorithm. The first marks the pointer *p->next* as deleted, changing its value but in such a way that the original pointer can still be reconstructed. The second actually deletes the node by resetting *p->next* as described in [8].

An example for a complicated case of node deletion shown in 4. When we delete node *10*, and node *20* is inserted at the meantime, then *20* will be lost, because the *CAS* connecting *head* to *30* cannot detect the change between *10* and*30* . Harris's Algorithm solved this issue with introducing logical deletion mark. It uses two *CAS* in the deletion of *10*, one for marking the next node of *10* as a mark of *10* beging deleted logically, another *CAS* for physical deletion. For the example case, at the insertion of *20*, it will detect from the marking on *30* that *10* is logically deleted, so it will invoke *search* to delete *10* physically before inserting *20* with *CAS* operation.

Figure 1, 2, 3 are the pseudo code of insert, remove and search operations in Harris' [5].

3 Methods

Due to the arbitrary access of element in the linked list, there is almost no locality, thus shared memory cannot be utilized. The only way to implement synchronization between arbitrary threads in a grid is through the atomic operations executing in global memory. Also, to support access to the arbitrary elements in the linked list, the linked list is kept in the global memory [6].

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CSC586B Project '24, Sept - Dec, 2024, Victoria, BC
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
public boolean List::insert (KeyType key) {
  Node *new_node = new Node(key);
  Node *right_node, *left_node;

  do {
    right_node = search (key, &left_node);
    if ((right_node != tail) && (right_node.key == key)) /*T1*/
      return false;
    new_node.next = right_node;
    if (CAS (&(left_node.next), right_node, new_node)) /*C2*/
      return true;
  } while (true); /*B3*/
}
```

Figure 1: List insert method attempts to insert a new node with supplied value [5]

```
public boolean List::delete (KeyType search_key) {
  Node *right_node, *right_node_next, *left_node;

  do {
    right_node = search (search_key, &left_node);
    if ((right_node == tail) || (right_node.key != search_key)) /*T1*/
      return false;
    right_node_next = right_node.next;
    if (!is_marked_reference(right_node_next))
      if (CAS (&(right_node.next), /*C3*/
          right_node_next, get_marked_reference (right_node_next)))
        break;
  } while (true); /*B4*/
  if (!CAS (&(left_node.next), right_node, right_node_next)) /*C4*/
    right_node = search (right_node.key, &left_node);
  return true;
}
```

Figure 2: List delete method attempts to delete a node with supplied value [5]

```
private Node *List::search (KeyType search_key, Node **left_node) {
  Node *left_node_next, *right_node;

search_again:
  do {
    Node *t = head;
    Node *t_next = head.next;

    /* 1: Find left_node and right_node */
    do {
      if (!is_marked_reference(t_next)) {
        (*left_node) = t;
        left_node_next = t_next;
      }
      t = get_unmarked_reference(t_next);
      if (t == tail) break;
      t_next = t.next;
    } while (is_marked_reference(t_next) || (t.key<search_key)); /*B1*/
    right_node = t;

    /* 2: Check nodes are adjacent */
    if (left_node_next == right_node)
      if ((right_node != tail) && is_marked_reference(right_node.next))
        goto search_again; /*G1*/
      else
        return right_node; /*R1*/

    /* 3: Remove one or more marked nodes */
    if (CAS (&(left_node.next), left_node_next, right_node)) /*C1*/
      if ((right_node != tail) && is_marked_reference(right_node.next))
        goto search_again; /*G2*/
      else
        return right_node; /*R2*/
  } while (true); /*B2*/
}
```

Figure 3: List search method finds the left and right nodes with supplied value [5]

Comparing to a sequential algorithm, the **overhead** of the parallel algorithm is at the stage of initializing the linked list on GPU, which includes:

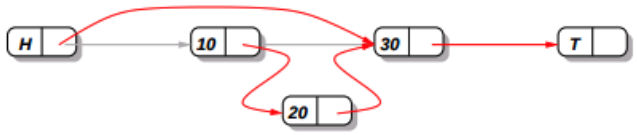

Figure 4: A Complicated Case of Node Deletion

- *ListSearch* for searching a node in the linked list, which cannot be paralleled.
- *ListTraversal* for deleting nodes at last.
- Allocation of global memory on GPU.
- Moving linked list data and operations data from CPU to GPU.
- Creation of linked list by adding nodes sequentially.

In our experiments comparing the performance with sequential algorithm, we are not counting the data allocation and data moving time. We only compare the massive arbitrary insertion and deletion operating time.

The **parallelism** is achieved by defining insertion, deletion, traversal, searching operations as global kernel functions, which support parallel execution for two or more threads, with each of the thread handles one operation for an arbitrary node.

3.1 Lock-free Data Structure Memory Management

We made some modifications to the original algorithm to tackle memory management issues. Memory management is one of the most cumbersome problems on lock-free data structures. In other words, when a thread removes a node from the structure, it cannot always free the memory for that node, because other threads might be holding a reference to this memory. Therefore, as a trade-off, on a remove operation the memory of the removed node will not be freed in our implementation, only deleted physically, which means *prev->next* points to *cur->next*. Additionally, this approach entails every insertion allocating a new node.

3.2 Logical Deletion Mark with Metadata Tagging

[5] proposed logical deletion mark in deletion. That means using double CAS for deletion, where one CAS is to change the pointer, another is to change logical deletion mark. We implemented the logical deletion mark with metadata tagging technique. The lowest three bits of a pointer's binary address is empty, so they can be used to store additional information. Thus, we use the lowest bit of the node pointer's address as a mark of being deleted logically, where 1 represents marked (deleted logically) and 0 represents unmarked (not deleted, still in the list). Specifically, the pointer address are converted to *unsigned long long* typed data to get their binary address, and three bitwise operators are defined for checking mark status or modifying the mark:

- *IS_MARKED*(p) Conduct ***bitwise AND*** operation with the pointer's binary address with *unsigned long long* 1, which is

Unpublished working draft. Not for distribution.

000...0001, then cast a *int* type conversion on it. The result is 1 if a list node pointer has been marked, otherwise 0.

- *GET_MARKED_REF*(p) Conduct ***bitwise OR*** operation with the pointer's binary address with *unsigned long long* 1. The result is the reference of the pointer with lowest bit set as 1, which means a node is marked. After a node is successfully deleted logically, the node pointer is marked with this operation.
- *GET_MARKED_REF*(p) Conduct ***bitwise NOT*** operation with the pointer's binary address with *unsigned long long* 1, which is *111...1110*. The result is the reference of the pointer with lowest bit set as 0, which means node mark removed. It is used for obtaining the actual address of any pointer, especially for a marked node.

The introducing of metadata tagging technique to mark the deleted nodes highlights its advantage in reducing memory usage and speedup the algorithm.

3.3 A Modification to Physical Deletion

In Harris' algorithm, the node is deleted physically (*prev->next* points to *cur->next*) in *delete* operation. It first tries to use CAS to modify the pointer, if failed, it will call search operation and delete the node in search as in Figure 2. In practice [4] [3], *search* is invoked by other *insert* and *delete* operations, as theses operations include searching for the target node at the beginning, rather than invoked everytime inside *delete* after logically deletion. We understand the reason of doing this, as in our experiments, invoking *search* inside *delete* significantly slows down the algorithm. However, it cannot guarantee all logically deleted nodes are physically deleted, since invoking *search* later is not guaranteed to traverse the whole list. Therefore, we add a traverse delete operation after all insertion and deletions, in which the marked nodes are deleted in the traversal. This can guarantee all logically deleted nodes are deleted physically at the end of all operations. Our approach can reach a better balance between performance and memory management.

Below is the added kernel function *listTraverseDel* to delete marked nodes physically.

```
__device__ void listTraverseDel()
{
    struct node *cur, *prev, *p, *prev_next; prev=head;
     cur=head->next;
     for(cur=head->next; cur->next; cur=(struct node*)
         GET_UNMARKED_REF(cur->next))
     {
         if(IS_MARKED(cur->next))
             continue;
         if(prev->next!=cur)
             prev->next=cur;
         prev=cur;
     }
}
```

4 Experiments

We conducted experiments to analyze the speedup with different problem size, the scalability and overhead of the parallel program, and the collision while using atomic operations.

4.1 Environment Setup

GPU settings: NVIDIA GeForce RTX 3090 with compute capability 8.6.

4.2 Overall Performance

In this experiment, we measure the speedup of conducting a set of insertion and deletion operations comparing to sequential version. To change the problem size, there are two parameters, one is N_1, the number of nodes in the initial linked list, the other is N_2, the number of operations. The operations (insert or remove) and the operating nodes are all generated randomly. Check Appendix A for data format.

We want to have as much parallelism as possible to see when it achieves maximum speedup, so we try to uses many threads as possible, we set the number of blocks to 32, the number of threads to 1024.

4.2.1 Experiment 1. Fix the number of initial list nodes to 10000, modify the number of operations in {100, 1000, 10000, 100000, 100000}, to see when it gets the highest speedup.

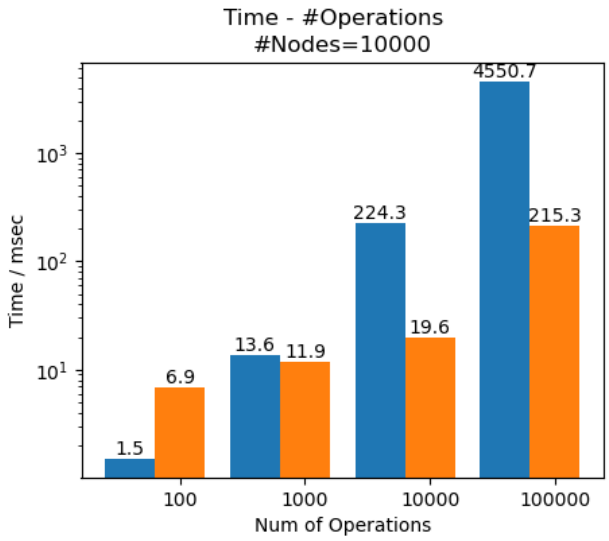

Figure 5: Time of Experiment 1

N_2 (Num of Operations)	Sequential Time / ms	Parallel Time / ms
100	1.5	6.9
1,000	13.6	11.9
10,000	224.3	19.6
100,000	4550.7	215.3

Table 1: Time Measurement of Experiment 1

From Figure 7 we can conclude that the number of operations has a significant impact on speedup. It needs to be large enough to have a significant speedup. When the number of operations is 100000 and the number of list nodes is 10000, it has 68x speedup compared

Unpublished working draft. Not for distribution.

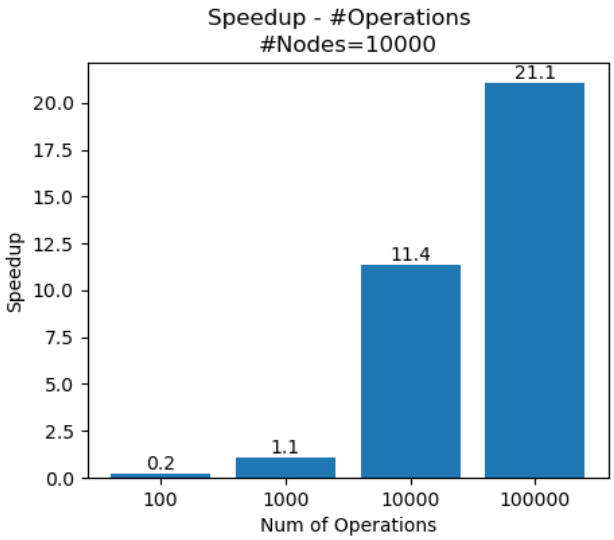

Figure 6: Speedup of Experiment 1

to the benchmark on CPU. The reason is that the traversal of the list takes equivalent time for both GPU and CPU, the insertion/deletion can be executed in parallel. In each insertion or deletion, it traverses the list to search for the target node, so the larger is the number of operations compared to the number of list nodes, the higher the speedup is.

4.2.2 Experiment 2. Fix the number of operations to 10000, modify the number of list nodes in {100, 1000, 10000, 20000, 30000, 40000}, to see when it gets the highest speedup.

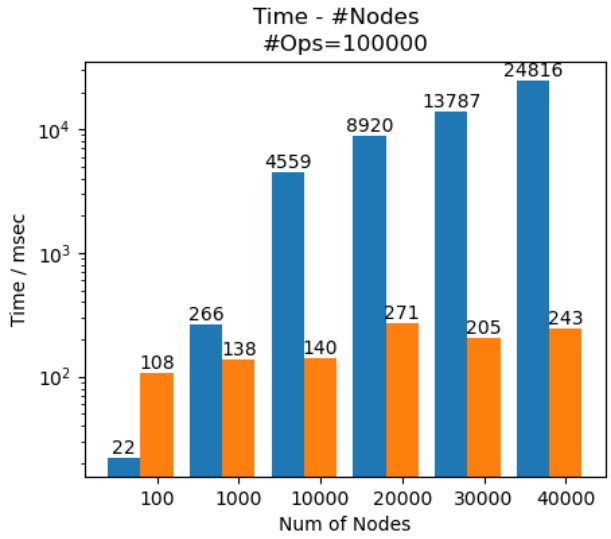

Figure 7: Time of Experiment 2

From Figure 8 we can conclude that the number of nodes need to be large enough to have significant speedup. When the number of list nodes is 40000, with the number of operations fixed to 100000, it has the highest speedup 141x compared to the benchmark on CPU. For N_1=50000 or larger, the GPU version cannot execute correctly. The reason is that the algorithm does not release the memory of deleted nodes, which causes memory leakage.

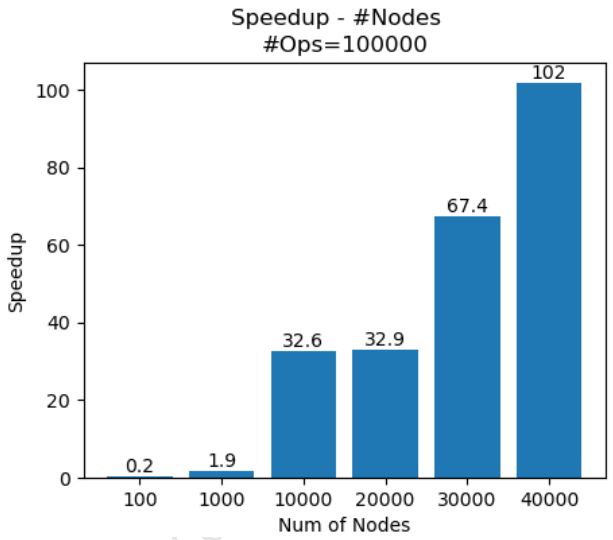

Figure 8: Speedup of Experiment 2

N_1 (Num of List Nodes)	Sequential Time / ms	Parallel Time / ms
100	22.3	107.5
1,000	265.9	138.5
10,000	4559.0	139.9
20,000	8919.7	271.2
30,000	13786.9	204.6
40,000	24816.4	243.3

Table 2: Time Measurement of Experiment 2

4.3 Scalability

In this experiment, we analyze the scalability of the parallel algorithm. We fix the number of nodes to 10000 and number of operations to 10000, the number of blocks ranges from {1, 2, 4, 8, 16}, the number threads per block ranges from {1,2,4,8,16,32,64,128,256,512}.

From Figure 9 we can conclude that it can speedup as the number of threads increases. Only when the threads is more than 512 it is faster than sequential algorithm.

4.4 Overhead Analysis

In this experiment, we analyzed the GPU and CPU performance, accounting for overhead work on a list of 10,000 nodes. The results are shown in Figures 10 From these, we see that overhead remains higher than the kernel time for the gpu under 1e5 operations. The cpu overhead stays low for the same range with the kernel time increasing greatly around 1e5 operations.

Unpublished working draft. Not for distribution.

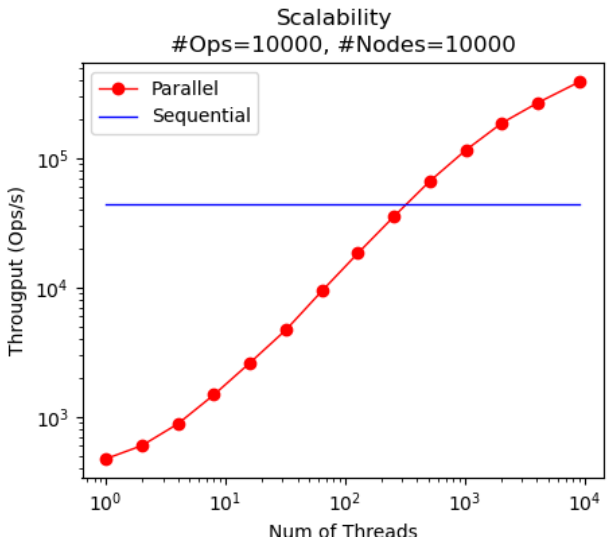

Figure 9: Scalability

Num of Total Threads	Time / ms	Throughput / Ops/s
1	21077.2	474
2	16640.5	600
4	11303.2	884
8	6747.5	1482
16	3818.6	2618
32	2125.5	4704
64	1062.4	9412
128	541.8	18456
256	281.8	35486
512	150.0	66666
1024 (2*512)	86.3	115874
2048 (4*512)	53.3	187617
4096 (8*512)	37.13	269323
9192 (16*512)	25.5	392156
1 Thread Sequential	224.3	44583

Table 3: Scalability

5 Collision of Operations by Different Threads

In this experiment, we analyze the collisions of operations by different threads. If the collision happens, the operation is unsuccessful and need to do again, which may costs extra time. Specifically, we conduct experiments to measure the collisions happening in two possible circumstances:

(1) **In *delete* operation**. After assigning right node variable with the next node of current node, we use CUDA *atomicCAS* to set actual *cur->next* as marked right node. If this actual right node is not the same as the assigned value before this atomic operation, that means other nodes are inserted meantime. Then we need to abort the deletion attempt, and search for the target node again.

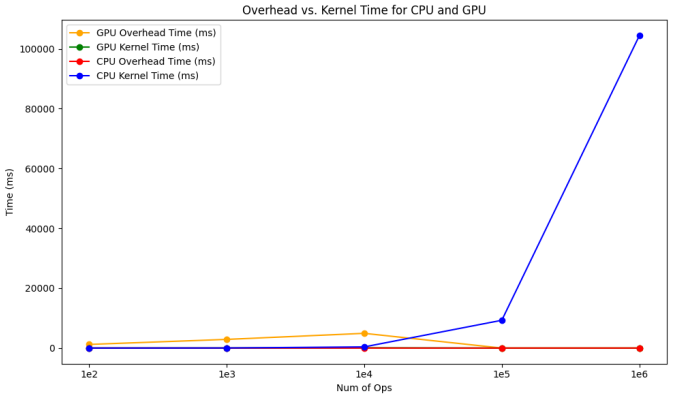

Figure 10: Overhead vs. Kernel Time for CPU and GPU

When the actual right is the same node as the right node, it means the logical deletion is completed correctly.

(2) **In *search* operation**. We delete marked nodes between the left and the current node. Loop from left to current node, if there is any unmarked node, it is inserted meantime, so we neet to abort the deletion attempt and search again, until no unmarked nodes in between. After that, we try to physically delete the marked nodes between left and right.

In the experiment, we fix the number of operations to 1000, the number of nodes to 1000. We change number of threads from 1 to 4096. From Figure 11 we can see, when the number of thread is 1, there is no collision as expected, since the algorithm becomes sequential. When the number of threads is fewer than numer of operations, the number of collisions increase with the increase of number of threads. When there are more threads than operations, the collision does not increase significantly. The ratio of the number of collisions / the number of active threads is around 10%. The number of active threads is $min(\#threads, \#ops)$. That means the algorithm and our implementation is effective.

Num of Total Threads	Time / msec	Collisions / 1000 Ops
Sequential	1.7	-
1	109.1	0
16	28.2	2
128	5.2	15
512	2.4	56
2*512	1.6	109
4*512	1.8	97
8*512	1.7	113

Table 4: Collisions of Operations

5.0.1 *Experiment 2.*

6 Conclusion and Future Work

We implemented Harris' algorithm with CUDA correctly, and evaluated it in comparison with sequential implementation. There are

Unpublished working draft. Not for distribution.

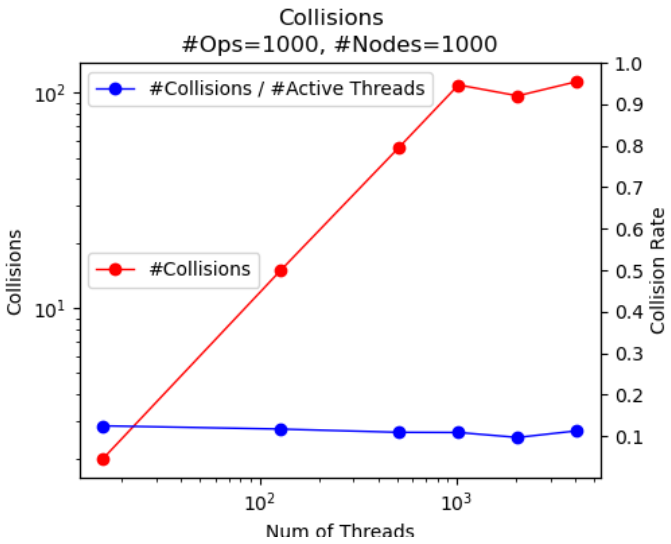

Figure 11: Collisions of Operations

still space for improvement, yet to be addressed. We need to tackle garbage collection in the future. Furthermore, several approaches can be also be investigated to mitigate the overhead especially the memory transfers and using an optimal grid configuration.

References

[1] [n. d.]. . https://www.cse.wustl.edu/~angelee/archive/cse539/spr15/lectures/lists.pdf

[2] [n. d.]. . https://github.com/xDarkLemon/concurrent_linked_list_cuda

[3] [n. d.]. . https://github.com/adolfoarangel/LockFreeLinkedList

[4] [n. d.]. . https://github.com/gavra0/ConcLinkedList

[5] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[6] Prabhakar Misra and Mainak Chaudhuri. 2012. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 53–60. https://doi.org/10.1109/ICPADS.2012.18

[7] John D. Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 214–222. https://doi.org/10.1145/224964.224988

[8] Wikipedia contributors. 2024. *Non-blocking linked list — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Non-blocking_linked_list Accessed: 2024-10-13.

[9] Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael F. Spear. 2013. Practical Non-blocking Unordered Lists. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8205)*, Yehuda Afek (Ed.). Springer, 239–253. https://doi.org/10.1007/978-3-642-41527-2_17

A Data Format

The operations are all generated randomly. For the initial linked list, each node has a unique value. The inserting nodes have unique values not duplicated with initial list nodes. The removing nodes are all from the initial linked list, not from the inserted nodes.
listnodes.txt: including initial linked list node values

```
5      // The number of nodes 5
4      // The node value
3
2 1
```

operations.txt: including operations to be conducted in parallel

```
3       // The number of operations
1 2 6   // <insert> <target node value> <insert value>
0 1     // <remove> <target node value>
1 3 8
```

B Library API

Note: head is defined on device, inaccessible from the host. The APIs are global kernel functions.

`__global__ void listInit()`

Initialize a linked list, create a head node, with node->data equal to -1 and node->next pointing to tail, and a tail node, with node->data equal to -1 and node->next pointing to NULL.

`__global__ void addFront(int val)`

Create a node with the given value and add it after the head node.

`__global__ void addFront(int *arr, int N)`

Create a sequence of nodes and add the after head node sequentially.

`__global__ void listPrint()`

Print the linked list from head to tail, and print the number of nodes.

`__global__ void listPrintLen()`

Print the list length.

`__global__ void listPrintRaw()`

Print all nodes in the list, including those marked as deleted logically but not deleted physically yet.

`__global__ void listTraverse()`

It is a wrapper of `__device__ void listTraverseDel()`. Traverse the linked list and delete all the marked nodes physically (modify the pointers pointing to them).

`__global__ void listInsert(int *ops, int *insertVals, int *insertPrevs, int N)`

Operate insertions in parallel.

`__global__ void listRemove(int *ops, int *Vals, int N)`

Operate deletions in parallel. Remove the nodes logically, marking them as deleted.

Received 12 Oct 2024; accepted 12 Oct 2024

Unpublished working draft. Not for distribution.