Lock-Free Linked List Library for GPUs

Final Project Report for NYU GPUs 2022 Fall

Yibo Liu
yl6769@nyu.edu
Dec 2022

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, in which both insertion and deletion operations support concurrent use: two or more threads of execution are able to perform insertions and deletions without interfering with each other's work.

The algorithms for concurrent linked lists are with locks and without locks, most of which are implemented on multi-core systems, with Pthreads for example. On one hand, using locks can cause blocking of threads, which will decrease the performance. On the other hand, CUDA does not support locks originally, but supports *atomicCAS* instead, while *compare-and-swap (CAS)* is used in lock-free algorithms.

Therefore, in this project, we implemented a lock-free algorithm and tested on a random generated operations dataset, achieving a maximum of 141x acceleration.

2. Related Work

For lock-free concurrent linked list algorithms, Valois[1] 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[2] 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, Zhang[3] added a subsequent traversal of the list which does garbage collection of logically deleted nodes. In this project we implement the Harris' algorithm.

3. Algorithm

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 [2] and the slides [4]. 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 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`. [5]

Here is the pseudo code of insert, remove and search operations in Harris' [2]:

```
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*/
}
```

Fig 1. List insert method attempts to insert a new node with supplied value [2]

```
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;
}
```

Fig 2. List delete method attempts to delete a node with supplied value [2]

```
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*/
}
```

Fig 3. List search method finds the left and right nodes with supplied value [2]

4. Implementation

Here are some important details about the implementation of this project.

4.1 Memory management

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, in this project, on a remove operation the memory of the removed node will not be freed, only deleted physically, which means `prev->next` points to `cur->next`. Additionally, this approach entails every insertion allocating a new node.

4.2 Logical deletion mark

The lowest three bits of a pointer can be used to store additional information, thus, in this project, the mark is stored in the lowest bit of the node pointer, where `1` represents marked and `0` represents unmarked. Bitwise operations are used to set mark and check status. This is a clever design, which uses minimal resources to store the mark.

4.3 A modification to Harris' remove operation

In Harris' algorithm, the node is deleted physically (`prev->next` points to `cur->next`) in remove operation. It first tries to use CAS to modify the pointer, if failed, it will call search operation and delete the node in search. However, in the experiments, this significantly slows down the algorithm, the reason might be when it calls search, there is an additional traverse to the list. Therefore, I do not physically delete the node in remove operation, and add a traverse delete operation after all insertion and deletions, in which the marked nodes are deleted in one traversal to the list. To clarify, not all deletions are done in traverse delete operation, some deletions are done in search operation called by insert or remove operations.

```
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;
}
```

Fig 4. The removed part of code of Harris' deletion method

```
__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;
        }
}
```

Fig 5. The kernel function listTraverseDel to delete marked nodes

5. Performance

5.1 Evaluation

We measure the speedup of conducting a set of insertion and deletion operations on GPU compared to CPU.

In the experiments, there are two parameters, one is `N1`, the number of nodes in the initial linked list, the other is `N2`, the number of operations. The operations (insert / remove) and the operating nodes are all generated randomly. Check **section 7.1** for test data format.

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.

5.2 Settings

All experiments are conducted on `cuda2` on NYU CIMS cluster.

CPU settings: AMD EPYC Processor (with IBPB). The experiment uses a single thread.

GPU settings: NVIDIA GeForce RTX 2080 Ti with compute capability 7.5. Total global memory is 11283904 KB. In the experiment, the number of threads is set to 1024, the number of blocks is `ceil(#operations/1024)`.

5.3 Experiments Result

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.

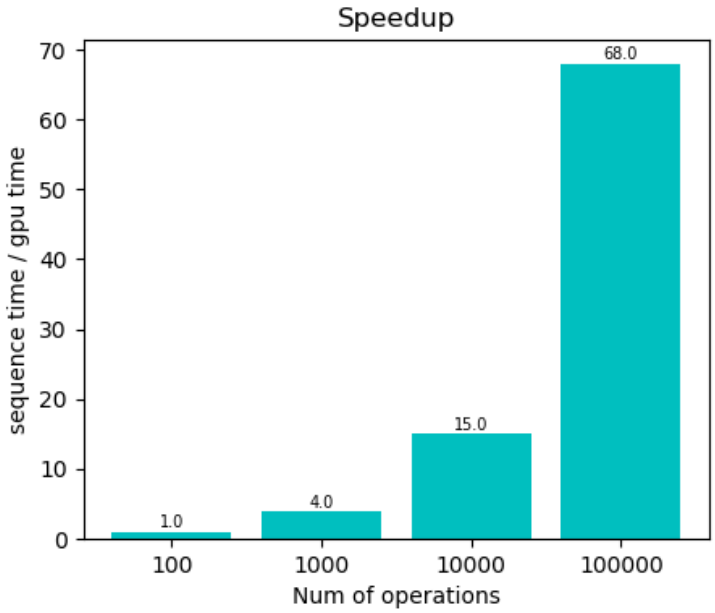

Fig 6. Speedup of experiment 1

N2 (Num of operations)	CPU Time / ns	GPU Time / ns
100	6,533,610	8,237,745
1000	58,004,314	14,709,008
10000	367,845,485	24,604,517
100000	8,330,191,845	123,235,450

Table 1. Time measurement of experiment 1

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

Experiment 2

Fix the number of operations to `100000`, modify the number of operations in `{100, 1000, 10000, 20000, 30000, 40000}`, to see when it gets the highest speedup. It is 10 times the number of list nodes.

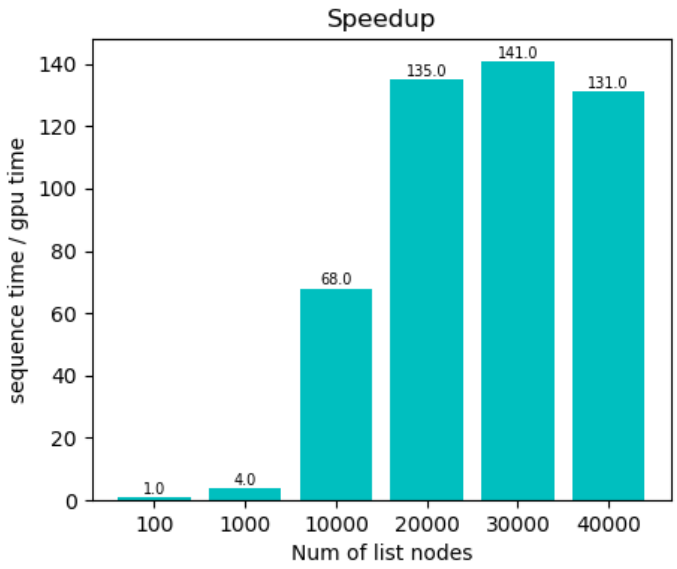

Fig 7. Speedup of experiment 2

N1 (Num of list nodes)	CPU Time / ns	GPU Time / ns
100	62,992,229	96,186,119
1000	326,486,847	86,162,376
10000	8,330,191,845	123,235,450
20000	21,344,506,702	157,606,459
30000	29,093,455,597	206,915,855
40000	33,613,348,792	256,001,324

Table 2. Time measurement of experiment 2

From the above figure we can conclude that the number of nodes need to be large enough to have significant speedup. When the number of list nodes is 30000, with the number of operations fixed to 100000, it has the highest speedup **141x** compared to the benchmark on CPU. For `N1=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.

6. Discussion

Though this library reaches a maximum of 141x acceleration on the insertion and deletion operations on the linked list, it has limitations due to its parallelism. It cannot guarantee order, so this library should be used in the situation that a lot of insertions and deletions happen to an existing long linked list. If the user wants to guarantee that the insertions are all successfully conducted, he should call the parallel insertion kernel first, then call `cudaDeviceSynchronize()`, then call the parallel deletion kernel. In this way, all deletions are executed after all insertions are done, ensuring the insertions are all successed.

For further improvements, the library can add garbage collection as Zhang[3] proposed to better manage the memory.

7. Library API and Usage

The library is available at https://github.com/xDarkLemon/Lock_Free_Linked_List_GPU.

7.1 APIs

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 listSearchOne(int val, struct node **p)
```

It is a wrapper of `__device__ struct node *listSearch(int val)`. Search the node with the given value and return.

```
__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.

7.2 Test Data Format

listnodes.txt: including initial linked list node values

```
5  // The number of nodes
5  // The node value
4
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
```

7.3 Command

Compile the GPU version with `nvcc -o gpull gpuLL.cu`

Compile the CPU version with `g++ -o cpull cpuLL.cpp`

Generate test data with `python3 gendata.py -n1 <number of initial nodes> -n2 <number of operations>`

The generated data will be saved to the directory `data/`.

Run `./gpull` and `./cpull` to get the result on the generated data.

8. References

[1] Valois, J. (1995), *Lock-Free Linked Lists Using Compare-and-Swap*, PODC '95 Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, pp. 214–222

[2] Harris, T. (2001), *A Pragmatic Implementation of Non-Blocking Linked Lists*, DISC '01 Proceedings of the 15th International Conference on Distributed Computing, Springer-Verlag London, UK, pp. 300–314

[3] Zhang, K.; Zhao, Y.; Yang, Y.; Liu, Y.; Spear, M. (2013), *Practical Non-Blocking Unordered Lists*, DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel

[4] https://cs.nyu.edu/courses/fall05/G22.2631-001/lists.slides2.pdf

[5] https://en.wikipedia.org/wiki/Non-blocking_linked_list