Lock-Free Linked-List Stack using CAS for GPU

Implementation Report

Aditya Azad (241110002) Sparsh Sehgal (241110071) Vraj Patel (241110080)

April 25, 2025

Abstract

This report details our implementation of a lock-free concurrent stack for GPUs based on a linked-list structure and Compare-and-Swap (CAS) atomic operations. This approach uses dynamically linked nodes allocated from a pre-defined memory pool. Key features: tagged pointer scheme using 64-bit atomics to prevent the ABA problem and a concurrent free list for efficient node reclamation and reuse. We have analyzed the core operations (push and pop) for performance characteristics, including scalability results from experimental tests with varying pool capacities.

Contents

1 Introduction

Concurrent data structures are essential for leveraging the massive parallelism offered by modern GPUs. While array-based structures like the Scan Stack offer advantages in memory access patterns, linked-list based structures provide flexibility in dynamic memory management. The lock-free linked-list stack, typically implemented using atomic Compare-and-Swap (CAS) operations, is a classic concurrent data structure.

The core aspects of our implementation are:

- A linked-list structure built from a node pool allocated in GPU global memory.
- Lock-free push and pop operations using 64-bit atomic CAS on the stack's top pointer.
- A tagged pointer scheme (combining a 32-bit node index and a 32-bit tag into a 64-bit atomic value) to mitigate the ABA problem.
- A concurrent free list, also managed with tagged pointers, for recycling stack nodes without kernel-wide synchronization.

This implementation provides a fundamental building block for GPU applications requiring LIFO data access with high concurrency.

2 Base Structure

The stack relies on a pool of nodes allocated in GPU global memory and atomic pointers to manage the stack top and the list of free nodes.

2.1 Node and Stack Structure Definition

The basic unit is a node containing data and a pointer to the next node. The stack structure holds pointers to the node pool, manages allocation, and maintains the atomic top pointers.

```
// Represents a node in the linked list
typedef struct Node {
    int data;
    uint32_t next_idx; // Index of the next node in the pool
} Node;

// Represents the index and tag for ABA prevention
typedef struct {
    uint32_t node_idx;
    uint32_t tag;
} TaggedIndex;

// Represents the stack structure in GPU memory
typedef struct Stack {
    Node* node_pool;         // Pointer to the pre-allocated node pool
    int   pool_capacity;     // Max number of nodes in the pool
    cuda::atomic<int, cuda::thread_scope_device> next_free_node_idx; // Index for initial node allocation
    cuda::atomic<uint64_t, cuda::thread_scope_device> stack_top;    // Atomic tagged pointer to stack top
    cuda::atomic<uint64_t, cuda::thread_scope_device> free_list_top; // Atomic tagged pointer to free list top
} Stack;

```

```
// Special index value for null pointers
#define NULL_INDEX UINT32_MAX
```

2.2 Tagged Pointers and 64-bit Atomics: Solving the ABA Problem

A big challenge in designing lock-free data structures using Compare-and-Swap (CAS) is the ABA problem. it can potentially corrupt the stack structure because the underlying state represented by A has changed.

To prevent this, our implementation employs a "tagged pointer" technique, efficiently utilizing a single 64-bit atomic variable (`cuda::atomic<uint64_t>`) to store both the pointer (node index relative to starting of node pool) and an ABA counter (tag). Here's how it works:

1. **Structure Packing:** The 64-bit unsigned integer is partitioned. The lower 32 bits store the actual `node_idx` (an index inside the `node_pool`, effectively acting as a pointer relative to start of pool). The upper 32 bits store a `tag`. This packing is achieved using bitwise operations, specifically left-shifting the tag and OR-ing it with the index, as shown in the `taggedIndexToAtomic` function. The reverse unpacking uses bitwise AND and right-shifting (`atomicToTaggedIndex`).

2. **Tag Increment on Reuse:** whenever a node is removed from the main stack and added to the `free_list_top` for recycling (during the `pop_gpu` operation), its associated `tag` is incremented (`next_tag = current_tag + 1`). This means every time a node index logically re-enters the system via the free list, it carries a new, unique tag.

3. **Single Atomic CAS Check:** Modern GPUs provides atomic CAS operations that work natively on standard integer sizes like 64 bits. By packing both the `node_idx` and the `tag` into a single `uint64_t`, we can use a single `compare_exchange_weak` operation provided by `cuda::atomic<uint64_t>`. This atomic primitive compares the *entire* 64-bit value (both index and tag simultaneously) that a thread expects to find in memory against the current 64-bit value.

4. **ABA Prevention:** If the ABA scenario occurs (A is popped and potentially the same index A is pushed back), the node A being pushed back will have an *incremented tag*. When the preempted thread T1 resumes, its expected 64-bit value (containing the original index A and the *old tag*) will no longer match the current 64-bit value in memory (which contains index A but the *new tag*). The single 64-bit CAS operation will therefore fail, correctly identifying that the state has changed and preventing the erroneous update.

```
// Combine TaggedIndex into a 64-bit value for atomic operations
// Shifts the tag to the upper 32 bits and ORs it with the index in
    the lower 32 bits
__host__ __device__ inline uint64_t taggedIndexToAtomic(TaggedIndex
    ti) {
    return ((uint64_t)ti.tag << 32) | (uint64_t)ti.node_idx;
}

// Extract TaggedIndex from a 64-bit atomic value
// Masks the lower 32 bits for the index, shifts right for the tag
__host__ __device__ inline TaggedIndex atomicToTaggedIndex(uint64_t
    val) {
    TaggedIndex ti;
    ti.node_idx = (uint32_t)(val & 0xFFFFFFFFULL); // Get lower 32 bits
    ti.tag      = (uint32_t)(val >> 32);           // Get upper 32 bits
    return ti;
}

// Helper to get Node* from index, with bounds checking
__device__ inline Node* indexToNodePtr(Stack* stack, uint32_t index) {
    if (index == NULL_INDEX) return nullptr;
    // Basic bounds check (could be more robust)
    if (index >= (uint32_t)stack->pool_capacity) return nullptr;
    return &stack->node_pool[index];
}
```

3 Stack Operations

The core push and pop operations rely on atomic CAS loops to modify the `stack_top` pointer in a lock-free manner. They interact with the free list for node allocation and reclamation.

3.1 Generic Tagged Stack Operations

Helper functions encapsulate the atomic CAS logic for manipulating tagged pointers, used by both the main stack and the free list. These functions use CUDA atomic memory orders like `memory_order_acquire` and `memory_order_release` to ensure correct visibility and ordering of memory operations across threads.

```
// Atomically pops a node index from a tagged list (stack or free list)
__device__ bool pop_tagged_stack(cuda::atomic<uint64_t, cuda::thread_scope_device>* list_top,
                                 TaggedIndex* result_ti, Stack* stack) {
    // Load the current 64-bit top value (index + tag)
    uint64_t old_top_atomic = list_top->load(cuda::memory_order_acquire);
    TaggedIndex old_top;
    TaggedIndex new_top;
    do {
        // Unpack the loaded 64-bit value
        old_top = atomicToTaggedIndex(old_top_atomic);
        if (old_top.node_idx == NULL_INDEX) return false; // List is empty

        Node* old_top_node_ptr = indexToNodePtr(stack, old_top.node_idx);
        // Handle potential invalid index or race condition if node ptr is bad
        if (!old_top_node_ptr) {
            // Re-read the atomic value if node pointer was bad and retry
            old_top_atomic = list_top->load(cuda::memory_order_acquire);
            continue;
        }

        // Prepare the new top pointer (next node index, same tag)
        new_top.node_idx = old_top_node_ptr->next_idx;
        new_top.tag = old_top.tag; // Tag does not change when popping

        // Attempt atomic CAS on the entire 64-bit value (index and tag)
        // If old_top_atomic matches the current memory value, update it to new_top's packed value
    } while (!list_top->compare_exchange_weak(old_top_atomic,
                                              taggedIndexToAtomic(new_top),
                                              cuda::memory_order_release,
                                              cuda::memory_order_acquire));
    // Success: return the popped tagged index (that was atomically read)
    *result_ti = atomicToTaggedIndex(old_top_atomic);
    return true;
}

// Atomically pushes a node index onto a tagged list
__device__ void push_tagged_stack(cuda::atomic<uint64_t, cuda::thread_scope_device>* list_top,
                                  uint32_t node_idx_to_push, uint32_t node_tag, Stack* stack) {
    uint64_t old_top_atomic = list_top->load(cuda::memory_order_relaxed);
    TaggedIndex old_top;
    TaggedIndex new_top;
    // Pack the new node's index and its current/new tag
    new_top.node_idx = node_idx_to_push;
    new_top.tag = node_tag;

    Node* node_to_push_ptr = indexToNodePtr(stack, node_idx_to_push);
    if (!node_to_push_ptr) return; // Invalid index

    do {
        // Unpack the current top to link the new node correctly
        old_top = atomicToTaggedIndex(old_top_atomic);
        node_to_push_ptr->next_idx = old_top.node_idx;

        // Attempt atomic CAS on the entire 64-bit value
        // If memory still holds old_top_atomic, update it to the packed new_top value
    } while (!list_top->compare_exchange_weak(old_top_atomic,
                                              taggedIndexToAtomic(new_top),
                                              cuda::memory_order_release,
                                              cuda::memory_order_acquire));
}
```

3.2 Push Operation (‘push_gpu‘)

The push operation acquires a node (either from the free list or the initial pool) and atomically links it to the top of the main stack.

Algorithm 1 GPU Lock-Free Stack Push Operation

```
1: Initialize node_idx_to_use to NULL_INDEX, node_tag to 0.
2: Try to pop a node from the free list: pop_tagged_stack(&stack->free_list_top, &reused_node_ti, stack)
3: if Pop from free list succeeded then
4:     Set node_idx_to_use = reused_node_ti.node_idx.
5:     Set node_tag = reused_node_ti.tag.                              ▷ Reuse tag from free list
6: else                                                                ▷ Free list was empty
7:     Atomically increment next_free_node_idx to get a new index: allocated_idx = stack->next_free_node_idx.fetch_add(1, cuda::memory_order_relaxed)
8:     if allocated_idx >= stack->pool_capacity then                  ▷ Node pool exhausted
9:         Decrement next_free_node_idx (best effort): fetch_sub(1, cuda::memory_order_relaxed)
10:        return false                                                ▷ Push failed
11:    end if
12:    Set node_idx_to_use = allocated_idx.
13:    Set node_tag = 0.                                               ▷ Initial tag for new node
14: end if
15: Get pointer to the node: node_to_use_ptr = indexToNodePtr(stack, node_idx_to_use)
16: if node_to_use_ptr is null then return false
17: end if                                                             ▷ Error
18: Set node_to_use_ptr->data = value.
19: Atomically push node onto main stack: push_tagged_stack(&stack->stack_top, node_idx_to_use, node_tag, stack)
20: return true                                                        ▷ Push successful
```

3.3 Pop Operation ('pop_gpu')

The pop operation atomically removes the top node from the main stack and then pushes it onto the free list with an incremented tag for reuse.

Algorithm 2 GPU Lock-Free Stack Pop Operation

```
1: Try to pop a node from the main stack: pop_tagged_stack(&stack->stack_top, &popped_ti, stack)
2: if Pop from main stack failed then                                 ▷ Stack is empty
3:     return false
4: end if
5: Get popped node index and tag: popped_node_idx = popped_ti.node_idx, current_tag = popped_ti.tag.
6: Get pointer to the popped node: popped_node_ptr = indexToNodePtr(stack, popped_node_idx)
7: if popped_node_ptr is null then return false
8: end if                                                              ▷ Error or race condition
9: Read data: *result = popped_node_ptr->data.
10: Calculate next tag for recycling: next_tag = current_tag + 1.     ▷ Increment tag for ABA prevention
11: Atomically push the node onto the free list: push_tagged_stack(&stack->free_list_top, popped_node_idx, next_tag, stack)
12: return true                                                        ▷ Pop successful
```

4 Memory Management and ABA Prevention

4.1 Node Pool and Free List

Memory for stack nodes is pre-allocated as a large array (`node_pool`) in GPU global memory. Initially, nodes are allocated sequentially using an atomic counter (`next_free_node_idx`). Once a node is popped from the main stack, instead of being truly freed, it is pushed onto a concurrent free list (`free_list_top`). Subsequent push operations first attempt to retrieve a node from this free list before allocating a new one from the pool. This recycling mechanism reduces the need for global synchronization or complex memory management schemes within the kernel.

4.2 ABA Problem Mitigation

As detailed in Section 2.2, the ABA problem is mitigated by packing a node index and an incrementing tag into a single 64-bit value. This allows a single atomic CAS operation to validate both the index and the tag simultaneously, ensuring that operations do not succeed based on stale information where the index has been reused with a different logical state.

5 Performance Analysis

5.1 Factors Affecting Performance

- **Contention Level:** The mix and rate of push/pop operations directly impact contention on the atomic pointers. The scalability tests show performance degradation under high concurrency.
- **Node Pool Size:** A sufficiently large `pool_capacity` is needed to avoid failures due to pool exhaustion. The scalability tests suggest pool size can interact complexly with performance, possibly due to caching and the effectiveness of the free list under different loads.
- **Cache Effects:** Performance can be influenced by how well the stack nodes and atomic pointers reside in the GPU's cache hierarchy. High contention can lead to cache line bouncing. The unusual scaling pattern with the smaller pool might be related to cache behavior.
- **Warp Scheduling:** The GPU's scheduling of warps can affect the interleaving of operations and the likelihood of CAS failures. Out test showed speedup of 30% to 40% when scheduled for warps compared to threads.

5.2 Scalability Test Results

The implementation was tested with an increasing number of operations, distributed across GPU threads using a fixed block size of 256. Two different node pool capacities were tested.

Table 1: Scalability Test Results (Block Size = 256, Pool Capacity = 102,400)

Test Name	Operations	Grid Size	Total Threads	Time (ms)	Throughput (MOps/s)
10k Ops	10,000	40	10,240	5.4221	1.84
50k Ops	50,000	196	50,176	80.6584	0.62
100k Ops	100,000	391	100,096	350.9781	0.28
200k Ops	200,000	782	200,192	2226.1174	0.09
500k Ops	500,000	1954	500,224	5616.1104	0.09
1M Ops	1,000,000	3907	1,000,192	14,253.8486	0.07

Table 2: Scalability Test Results (Block Size = 256, Pool Capacity = 10,240)

Test Name	Operations	Grid Size	Total Threads	Time (ms)	Throughput (MOps/s)
10k Ops	10,000	40	10,240	5.4282	1.84
50k Ops	50,000	196	50,176	15.3088	3.27
100k Ops	100,000	391	100,096	14.9442	6.69
200k Ops	200,000	782	200,192	15.3394	13.04
500k Ops	500,000	1954	500,224	641.9886	0.78
1M Ops	1,000,000	3907	1,000,192	1678.4343	0.60

Analysis: The results show contrasting trends depending on the pool capacity.

- With a large pool capacity (102 400, Table 1), throughput decreases monotonically and significantly as the number of operations (and concurrent threads) increases. This aligns with the expectation that contention on the atomic head pointers becomes the dominant bottleneck, limiting scalability.
- With a smaller pool capacity (10 240, Table 2), the behavior is peculiar. Throughput initially increases dramatically, peaking at 200k operations (13.04 MOps/s), before dropping sharply at 500k operations. This suggests that with a smaller pool, node recycling via the free list becomes crucial much earlier. The initial performance increase might be due to cache effects (better locality as nodes are rapidly recycled within the smaller pool) or perhaps an artifact of the workload interacting with the free list dynamics. However, as the number of operations further increases (500k and 1M), contention likely becomes overwhelming, or the free list itself becomes a bottleneck, causing throughput to plummet, although it remains higher than the large-pool case for the 500k and 1M tests.

Overall, the results highlight that while the lock-free approach works, its scalability on the GPU is limited by contention, and performance can be sensitive to factors like pool size and the resulting memory access/recycling patterns. The non-monotonic scaling with the smaller pool warrants further investigation.

6 Implementation Details and Challenges

6.1 64-bit Atomics

The reliance on `cuda::atomic<uint64_t>` with `cuda::thread_scope_device` is crucial for the correctness of the tagged pointer scheme, as detailed in Section 2.2. This ensures atomicity across threads within the same GPU device for combined index and tag updates, efficiently preventing the ABA problem.

6.2 Concurrent Free List Management

Managing the free list concurrently using the same tagged atomic CAS mechanism as the main stack adds complexity but allows nodes to be recycled without requiring locks or global barriers. Handling potential errors like failing to get a node pointer (`indexToNodePtr`) requires careful checks within the CAS loops. The performance impact of contention on the free list pointer itself is another factor, especially evident in the small pool tests.

6.3 Pool Exhaustion

The implementation handles running out of nodes in the initial pool by checking the value returned by `next_free_node_idx.fetch_add`. If the pool is exhausted and the free list is empty, push operations will fail, as confirmed by Verification Test [3].

6.4 Practical Testing with Host-Generated Values

The verification test `run_verification_pops_gt_pushes` demonstrates a practical testing approach where known values are generated on the host CPU, copied to the GPU (using `d_predefined_push_value` pushed onto the stack by the kernel, and then popped. The results are copied back to the host and verified against the original values using standard library containers (`std::multiset`) to ensure correctness (both value and count). This provides stronger validation than just checking success flags and confirms the node reclamation works correctly.

7 Conclusion

This report described a GPU implementation of a lock-free concurrent stack using a linked list, Compare-and-Swap (CAS) operations, and a tagged pointer scheme. Key features include:

- Lock-free semantics via atomic CAS loops on tagged pointers.
- ABA problem prevention using the 64-bit index-tag combination, validated with a single atomic operation.
- Efficient node recycling through a concurrent free list.
- Demonstrated verification and scalability testing methodologies.

While functionally correct and capable of handling concurrency, the scalability results indicate that performance is heavily limited by contention on the shared atomic pointers (`stack_top` and `free_list_top`), especially with a large number of threads. The interaction between pool capacity, node recycling, and contention presents complex performance characteristics, as seen in the differing scaling trends between the large and small pool tests. This highlights the challenges of implementing highly scalable concurrent data structures on massively parallel architectures like GPUs using simple CAS-based linked lists.

The source code for this implementation is available on GitHub at:
https://github.com/Vrajb24/ScanStack-Implementation.git

To compile and run the CAS Stack code (based on the provided `main` *function and previous report context):*

```
# Assuming the code is in a file named CAS_stack.cu
nvcc -O2 --allow-unsupported-compiler -ccbin=g++-9 -std=c++17
    -arch=sm_86 -lineinfo -res-usage CAS_stack.cu -o cas_output &&
    ./cas_output
```

(Adjust `-ccbin`, `-arch=sm_XX` as needed for your compiler and GPU. For A40 use `-arch=sm_86`)

8 Future Work

Potential areas for further exploration include:

- *Implementing more sophisticated backoff strategies (e.g., exponential backoff) within the CAS loops to potentially reduce contention under heavy load, potentially improving scalability, especially in the high-contention regime seen with the large pool.*
- *Investigating the non-monotonic scaling behavior observed with the smaller pool capacity through detailed profiling (e.g., measuring cache misses, atomic operation success/failure rates).*
- *Exploring dynamic resizing of the node pool if pre-allocation is insufficient or wasteful for certain applications.*

- *Comparing performance against other GPU concurrent stack implementations (e.g., the Scan Stack, library implementations) under various workloads, especially focusing on high-contention scenarios and different pool sizes.*

References

[1] Herlihy, M., Shavit, N. (2012). The Art of Multiprocessor Programming. Morgan Kaufmann. (General reference for lock-free data structures and ABA problem)

[2] NVIDIA CUDA C++ Programming Guide. Atomic Functions. Accessed April 25, 2025. (Reference for `cuda::atomic` usage)

[3] Michael, M. M., Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (pp. 267-275). (Classic paper on lock-free structures using CAS, relevant principles)

[4] Hendler, D., Shavit, N., Yerushalmi, L. (2004). A scalable lock-free stack algorithm. In Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (pp. 206-215). (Relevant work on CPU lock-free stacks)