Graphics Processing Units (GPUs): Architecture & Programming Project

Project Group No.: 12
Project Title: Deque Data Structure Library for GPUs

Project Group Members:

1. **Name**: Darshan Dinesh Kumar
   NetID: dd3888
   University ID: N10942768

Abstract

The end of Moore's Law necessitates the development of innovative solutions to augment the performance of applications rather than attempting to pack more transistors on a chip and/or increasing the CPU frequency. In this direction, Graphics Processing Units (GPUs) are now ubiquitous for high-performance computing and enable a wide variety of applications. However, programming for GPUs is not necessarily straightforward nor intuitive. It requires a fundamental shift in thinking by the Software Developers. Consequently, this new paradigm and programming model can hinder the adoption of GPUs despite their immense capabilities. Further, even when adopted, they may be severely under-utilized due to inefficiencies in the developed Software. Building and supporting a data structure library for GPUs can significantly bridge this gap. This project is precisely such an effort to design and implement a data structure library for GPUs. The project specifically implements a generic (supporting all data-types) deque (double-ended queue) data structure which is a fundamental building block for a variety of applications and is highly versatile given that it can be reduced to represent other structures like stacks and queues. As part of this project, various implementations of the deque were experimented with and analyzed, including a naïve global lock-based implementation, a lock-free atomic counter-based implementation, both for global use across different blocks of a grid, and a shared memory-based implementation for intra-block communication. The generated results for the various benchmarks highlight the pros and cons of the different implementations and denote their suitability for different kinds of applications. The results for the lock-free version are quite promising as compared to the production grade, stdgpu library's deque, with an approximate 36% improvement in execution time for a benchmark. Further, the block-level shared memory deque demonstrates a significant improvement, of up to 67% as compared to the lock free version, highlighting its suitability for applications requiring a deque only for Intra-Block coordination.

Introduction

For decades, the computer industry benefited from Moore's Law, which predicted a steady rise in transistor density and single-core CPU performance. This trend has now slowed due to physical limits like heat (the "power wall") and manufacturing costs. This end of "free"

performance gains has forced a major shift from improving single-thread speed to developing massively parallel processors.

Graphics Processing Units (GPUs) are a key part of this new landscape. Originally for graphics, they are now widely used in high-performance computing (HPC), AI, and data analytics. Their architecture, with thousands of cores, is ideal for parallel tasks, enabling massive speedups for many applications.

However, using this power is not easy. Programming a GPU is very different from programming a CPU. It requires a complete shift in thinking, as developers must learn new programming models like CUDA or OpenCL and manage complex memory and execution systems to handle massive parallelism. Developers must manage data across a grid of thread blocks, move data between fast (shared) and slow (global) memory, and handle synchronization to prevent errors in an environment with thousands of running threads. This steep learning curve is a major barrier for many developers, slowing down the adoption of GPUs.

Even when developers learn this new model, their applications are often under-utilized. A simple, "naive" port of a C.P.U. algorithm to a G.P.U. might even run *slower* due to data bottlenecks or inefficient resource use. Achieving real performance requires expert-level optimization and fundamental changes to the software's design.

One way to bridge this gap is with high-level libraries that provide familiar tools. A good library hides the complex, low-level details of parallel programming and offers a simple API, much like the C++ Standard Template Library (STL) does for CPUs. This project is an effort to create such a data structure library for GPUs.

This work specifically implements the **deque** (double-ended queue), a very versatile and fundamental data structure. A deque allows efficient insertion and deletion of items from both its front and back. This flexibility means it can be used as a stack (last-in, first-out) or a queue (first-in, first-out). In parallel computing, these structures are key building blocks for task schedulers, work queues, and graph algorithms.

As part of the research and exploration under this project, various implementations of the deque were experimented with and analyzed. This included an initial naïve lock-based implementation that exposes the typical interfaces of a deque but supports them by internally using an expensive global spinlock through atomic CAS (Compare And Swap) on the deque. The second version attempts to build on top of the first one and overcome the overheads due to the global lock by using atomic counters on the buffer slots, thereby supporting a global lock-free deque. This version significantly outperforms the first version. The third implementation is based on the lock-free version but it houses the deque entirely in the shared memory, thereby targeting only intra-block communication. Various functional and performance benchmarks were developed to verify the functionality and compare performance of the different implementations for different applications. Further, benchmarks were also written to compare these versions with the deque supported by stdgpu, an open-source library providing generic GPU data structures for fast and reliable data management. The results for the lock-free version are quite promising as compared to

the production grade stdgpu library's deque, with an approximate 36% improvement in execution time for the LIFO benchmark. Further, the block-level shared memory deque demonstrates a significant improvement, of up to 67% as compared to the lock free version, highlighting its suitability for applications requiring a deque only for Intra-Block coordination.

Literature Survey

The challenge of making GPU hardware more accessible to developers is not new, and existing solutions provide a valuable foundation for this project. The body of relevant work can be broadly classified into three categories: (1) high-level abstraction libraries that parallelize standard interfaces, (2) low-level performance primitives that act as building blocks, and (3) academic research into the design of concurrent, parallel-safe data structures.

High-Level Abstraction Libraries

This category includes libraries designed to provide a C++ Standard Template Library (STL) or standard-library-like experience for GPU programming. The primary goal is to abstract away the complexities of the underlying programming model, such as kernel launches and memory management.

- **NVIDIA Thrust:** Thrust is the most prominent library in this space, bundled directly with the NVIDIA CUDA Toolkit. It provides a high-level C++ interface for data-parallel algorithms and containers.
    - **Pros:** Thrust is exceptionally well-optimized, robust, and widely adopted. It allows developers to write clean, expressive code using familiar concepts like thrust::vector, thrust::sort, and thrust::reduce. It cleverly uses a backend system to dispatch work to CUDA, OpenMP, or TBB, making code portable.
    - **Cons:** Thrust's design philosophy centers on data parallelism, not task-parallel concurrency. It lacks fundamental concurrent containers like a thrust::queue or thrust::deque.
- **stdgpu:** This is a more recent, C++17-based effort to provide a standard library interface for GPU and other accelerator programming. In fact, this is closely related to this project in that it supports data structures like deques.
    - **Pros:** It leverages modern C++ features and aims to provide an API that is even closer to the std:: namespace, promoting a single-source C++ standard.
    - **Cons:** Like Thrust, stdgpu's focus is on algorithms (stdgpu::sort) and data-parallel containers (stdgpu::vector).

Low-Level Performance Primitives

This category includes libraries that provide highly-optimized, but lower-level, building blocks. These are less for end-users and more for "expert" developers who are building higher-level libraries (like Thrust).

- **NVIDIA CUB:** The CUB library (now part of the CUDA Toolkit) provides high-performance, reusable "collective" primitives for threads within a single block.

    - **Pros:** CUB is the gold standard for intra-block performance. It provides lightning-fast implementations of operations like BlockScan, BlockReduce, and, most relevantly, BlockQueue. These primitives are essential for building efficient, larger-scale structures.
    - **Cons:** CUB is a "kit of parts," not a complete solution. The BlockQueue it provides, for example, is explicitly for communication *within* a block and operates on a fixed-size buffer in shared memory. It does not provide a grid-wide, deque for communication *between* blocks, which is a far more complex problem.

Academic Research on Concurrent Data Structures

This category covers research papers that analyze the core challenges of building parallel-safe data structures, both on multi-core CPUs and GPUs. While not libraries, they inform the implementation of this project.

- **CPU-based Concurrent Deques:** A vast body of work exists on lock-free data structures for multi-core CPUs. Foundational designs like the **Arora-Blumofe-Plaxton (ABP) deque** and the closely related **Chase-Lev deque** became the standard for high-performance work-stealing schedulers (used in systems like Intel's TBB, Cilk, and ParlayLib).
    - **Pros:** These algorithms are highly optimized for the specific CPU memory model and its common "work-stealing" pattern, which involves one "owner" thread pushing to and popping from one end (the "bottom") and multiple "stealer" threads taking work from the other end (the "top").
    - **Cons:** The architecture and memory model of a GPU are drastically different from a CPU. A CPU has a few, highly independent cores with deep caches. A GPU has thousands of "lightweight" threads executing in lockstep (warps). A direct port of an ABP or Chase-Lev deque will perform poorly due to issues like warp divergence, the different costs of atomic operations, and the lack of a coherent cache hierarchy in the same way a CPU has.
- **GPU-Specific Concurrent Queues:** Several researchers have specifically tackled the problem of high-throughput queues on GPUs. These works often explore lock-free implementations using atomic "compare-and-swap" (CAS) operations. Seminal work in this area, such as *Stuart and Owens' "A Scalable, Lock-Free FIFO Queue for GPUs"*, analyzed the significant challenges of contention when thousands of threads attempt to access a single head or tail pointer. Later research, like *Tzeng et al.'s "A Generic and Scalable Lock-Free FIFO Queue for GPUs"*, further refined these techniques to reduce atomic operations and improve throughput.
    - **Pros:** These papers provide a direct analysis of the trade-offs of using atomics on a GPU, managing contention from thousands of threads, and the overhead of device-wide memory synchronization. They establish the core techniques, such as using CAS loops and managing memory visibility, that are necessary for any concurrent GPU structure.
    - **Cons:** This research has a few key limitations relevant to our project:
        1. **Queue, not Deque:** The focus is almost exclusively on FIFO queues or LIFO stacks, not the more general-purpose, double-ended deque.

2. **Not a Library:** These are academic papers, presenting an algorithm and results. They are not robust, production-ready, or easy-to-use libraries that a developer can simply include.

The literature survey reveals a clear gap in the existing ecosystem. **High-level libraries (Thrust, stdgpu)** provide ease-of-use but lack the *specific concurrent structures* needed for dynamic task parallelism although stdgpu does support a deque. **Low-level primitives (CUB)** provide the *building blocks* (like an intra-block queue) but are not a complete, grid-wide solution and require expert knowledge. **Academic research** provides the *theory* but has focused on simpler structures (queues, not deques) or has not been productized into a usable library. This project is needed because it aims to fill this "gap in the middle."

Proposed Idea

As part of the research, experimentation and development of a generic Deque Data Structure Library for GPUs, 3 versions of the Deque Library were developed and experimented with as follows:

1. **Lock-based GPU Deque:**

   The source code for this version is contained within the *src/1_lock_based_deque* directory as a header file that can be included in client CUDA programs.
   This is a naïve implementation of the Deque data structure for GPUs using locks. This design is focused on simplicity and guaranteeing correctness. It serves as a way to benchmark and compare future implementations.

   The easiest way to make a data structure thread-safe is to protect it with a single, global lock. This ensures that only one thread can modify the deque's state (its front, rear, or count indices) at any given time. This mutual exclusion completely eliminates all race conditions.

   However, this simplicity comes at a high and obvious performance cost, especially on a GPU. By using a single lock, we serialize access. If thousands of threads from different blocks try to push or pop elements simultaneously, they will form a "virtual line," and all but one will be forced to wait. This effectively neutralizes the massive parallelism that the GPU is designed for. This implementation is, therefore, "naive" in a parallel context, but it serves as an essential and correct starting point.

   It has to be noted that this version (and all the other versions discussed later) of the Deque has been implemented as a generic data structure and hence can support any data type.

The implementation is split into two distinct components:

a. GpuDequeHandle<T> (Host-side): This is a C++ class that acts as a "controller" or "factory" living on the CPU. Its job is to manage the lifetime of the deque's resources.
    - Constructor: When we create a GpuDequeHandle(capacity), it performs two key allocations in global device memory:
        1. cudaMalloc for the buffer: This is the large array that will hold the actual data elements (eg., int, float, etc.).
        2. cudaMalloc for the GpuDeque<T> struct: This is a small allocation for the struct itself, which holds the deque's state (pointers, indices, and the mutex).
    - Initialization: It then initializes the state (e.g., front = 0, rear = 0, count = 0, mutex = 0) on the host and copies this initial state to the GpuDeque<T> struct on the device.
    - get_device_ptr(): This function returns the device-side pointer to the GpuDeque<T> struct. This is the handle that you pass as an argument to your CUDA kernel.
    - Destructor: When the handle goes out of scope on the host, it safely frees both the buffer and the GpuDeque<T> struct from device memory.
b. GpuDeque<T> (Device-side): This is a struct that contains only __device__ functions and the data state. This is the component that the threads inside the kernel will interact with.
   The device-side GpuDeque operates on two core principles:
    - Atomic Spinlock: The mutex is a simple int used as a lock.
        1. lock(): This function uses atomicCAS(&mutex, 0, 1). It attempts to atomically "Compare and Swap" the value of mutex. If mutex is 0 (unlocked), it swaps it to 1 (locked) and returns 0, exiting the while loop. If mutex is already 1, the atomicCAS fails (returns 1), and the thread "spins" in the while loop, repeatedly trying until it acquires the lock. This is a busy-wait and hence serializes the operations.
        2. unlock(): This function uses atomicExch(&mutex, 0) to atomically set the mutex value back to 0, releasing the lock and allowing another waiting thread to proceed.
    - Circular Buffer: The buffer is a fixed-size array, and the front and rear indices "wrap around" when they hit the end of the array. This is a classic and efficient way to implement a bounded queue or deque.
        1. The state is managed by front (index of the first element), rear (index of the next available slot after the last element), and count (current number of elements).
        2. The modulo operator (% capacity) is the key to this "wrap-around" logic. For example, (rear + 1) % capacity correctly increments the rear index, wrapping it from capacity-1 back to 0.

3. The count variable is used to distinguish between a full and empty state.

The following __device__ functions are exposed as interfaces to the deque. All operations (except getCapacity) acquire and release the global lock, making them atomic and thread-safe, but also serializing access.

__device__ bool pushFront(const T& element)

- **Arguments:** const T& element - The data element to add to the front of the deque.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    1. Acquires the lock().
    2. Checks if deque is full. If so, unlocks and returns false.
    3. Calculates the new front index: front = (front - 1 + capacity) % capacity.
    4. Writes the element to buffer[front].
    5. Increments count.
    6. Releases the lock through unlock().

__device__ bool pushBack(const T& element)

- **Arguments:** const T& element - The data element to add to the back of the deque.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    1. Acquires the lock().
    2. Checks if deque is full. If so, unlocks and returns false.
    3. Writes the element to buffer[rear].
    4. Calculates the new rear index: rear = (rear + 1) % capacity.
    5. Increments count.
    6. Releases the lock through unlock().

__device__ bool popFront(T* out_element)

- **Arguments:** T* out_element - A pointer to a variable where the popped element will be stored.
- **Return Value:** bool - true on success, false if the deque is empty.
- **Logic:**
    1. Acquires the lock().
    2. Checks if deque is empty. If so, unlocks and returns false.
    3. Copies the front element: *out_element = buffer[front].
    4. Calculates the new front index: front = (front + 1) % capacity.
    5. Decrements count.
    6. Releases the lock through unlock().

__device__ bool popBack(T* out_element)

- **Arguments:** T* out_element - A pointer to a variable where the popped element will be stored.

- **Return Value:** bool - true on success, false if the deque is empty.
- **Logic:**
    1. Acquires the lock().
    2. Checks if deque is empty. If so, unlocks and returns false.
    3. Calculates the index of the last element: rear = (rear - 1 + capacity) % capacity.
    4. Copies the rear element: *out_element = buffer[rear].
    5. Decrements count.
    6. Releases the lock through unlock().

__device__ bool isEmpty()

- **Logic:** Acquires the lock, returns (count == 0), and releases the lock.

__device__ int getSize()

- **Logic:** Acquires the lock, copies count to a local variable, releases the lock, and returns the copy.

__device__ int getCapacity() const

- **Logic:** Returns the capacity, a read-only value. Hence, this operation does not require a lock.

2. **Lock-Free GPU Deque:**

The source code for this version is contained within the *src/2_lock_free_deque* directory as a header file that can be included in client CUDA programs.
The primary and severe limitation of the naive lock-based implementation is the serialization of all accesses. A single global lock becomes a massive point of contention, effectively eliminating all parallelism and forcing thousands of threads to wait in line. This design is the bottleneck.

The major objective for this second design is to eliminate the bottleneck by moving to a lock-free approach. The goal is to allow multiple threads to attempt operations (pushes and pops) from both ends of the deque concurrently, without ever having to wait for a lock. This is achieved by using atomic operations not as locks, but as counters to "claim" a slot or an item. This should, in theory, scale significantly better with a high number of concurrent threads.

The high-level architecture remains identical to the lock-based version, which is a key advantage. The GpuDequeHandle<T> (host-side) class is almost exactly the same. It allocates the device buffer and the GpuDeque<T> struct, but instead of initializing a mutex, it initializes head = 0 and tail = 0. The get_device_ptr() and destructor logic are the same, demonstrating a clean separation of concerns between resource management (host) and concurrent logic (device).

The underlying implementation details of this lock-free design are completely

different from the naive version. It abandons front, rear, and count indices. Instead, it uses two unbounded atomic counters:

- int head: This counter tracks the "head" of the deque. It is atomically incremented by popFront and atomically decremented by pushFront. It logically counts the total items popped from front.
- int tail: This counter tracks the "tail" of the deque. It is atomically incremented by pushBack and atomically decremented by popBack. It logically counts the total items pushed to back.

In this model, the state of the deque is defined by the difference between these counters:

- Current Size: tail – head
- Empty: head >= tail
- Full: (tail - head) >= capacity

The core of every operation follows a "Claim-Check-Commit/Rollback" pattern:

- Claim: A thread first atomically claims its right to an item or slot by modifying one of the counters (e.g., atomicAdd(&tail, 1) for a pushBack). The value returned by the atomic operation (the value before the add) serves as the thread's unique "ticket."
- Map: This "ticket" (the unbounded counter value) is then mapped to an index in the bounded buffer using the positive_mod(ticket, capacity) function.
- Check: The thread then performs a check to see if the operation is valid.
    - For a push, it checks if the deque is full: if (ticket - other_counter >= capacity).
    - For a pop, it checks if the deque is empty: if (ticket >= other_counter).
- Commit or Rollback:
    - Commit: If the check passes, the operation is valid. The thread proceeds to write to or read from its claimed slot in the buffer (e.g., buffer[index] = element).
    - Rollback: If the check fails (full or empty), the operation is invalid. The thread atomically undoes its claim (e.g., atomicSub(&tail, 1)) and returns false.

Memory Visibility: A critical detail is the use of *(volatile int*)&head and *(volatile int*)&tail when performing the "Check." This volatile cast is a low-level technique to force the compiler to perform a fresh read from device memory, bypassing any value that might be cached in a register. This ensures the thread sees the most up-to-date value of the other counter as modified by other threads.

The following __device__ functions are exposed as interfaces to the deque. They allow concurrent access, with conflicts being resolved by atomic operations and the "Claim-Check-Rollback" logic as described above.

__device__ bool pushBack(const T& element)

- **Arguments:** const T& element - The data element to add to the back.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    - **Claim:** int my_tail = atomicAdd(&tail, 1); claims a "push" ticket. my_tail is the value before the increment, which will be this thread's slot.
    - **Check:** if (my_tail - (*(volatile int*)&head) >= capacity) checks if the newly claimed slot is "too far" ahead of the head, meaning the buffer is full.
    - **Rollback:** If full, atomicSub(&tail, 1); rolls back the claim. Returns false.
    - **Commit:** If not full, writes the element: buffer[positive_mod(my_tail, capacity)] = element;. Returns true.

__device__ bool popFront(T* out_element)

- **Arguments:** T* out_element - Pointer to store the popped element.
- **Return Value:** bool - true on success, false if the deque is empty.
- **Logic:**
    - **Claim:** int my_head = atomicAdd(&head, 1); claims a "pop" ticket. my_head is the value of the slot to be popped.
    - **Check:** if (my_head >= (*(volatile int*)&tail)) checks if the head has caught up to or passed the tail, meaning the deque is empty.
    - **Rollback:** If empty, atomicSub(&head, 1); rolls back. Returns false.
    - **Commit:** If not empty, reads the element: *out_element = buffer[positive_mod(my_head, capacity)];. Returns true.

__device__ bool pushFront(const T& element)

- **Arguments:** const T& element - The data element to add to the front.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    - **Claim:** int my_head = atomicSub(&head, 1); claims a "push front" ticket by decrementing the head.
    - **Check:** if ((*(volatile int*)&tail) - my_head >= capacity) checks if the tail is now "too far" from the new head.
    - **Rollback:** If full, atomicAdd(&head, 1); rolls back. Returns false.
    - **Commit:** If not full, writes the element: buffer[positive_mod(my_head - 1, capacity)] = element;. (It writes to my_head - 1 because my_head was the value after the subtraction).

__device__ bool popBack(T* out_element)

- **Arguments:** T* out_element - Pointer to store the popped element.
- **Return Value:** bool - true on success, false if the deque is empty.

- **Logic:**
    - **Claim:** int my_tail = atomicSub(&tail, 1); claims a "pop back" ticket by decrementing the tail.
    - **Check:** if ((*(volatile int*)&head) >= my_tail) checks if the head has caught up to the new, smaller tail.
    - **Rollback:** If empty, atomicAdd(&tail, 1); rolls back. Returns false.
    - **Commit:** If not empty, reads the element: *out_element = buffer[positive_mod(my_tail - 1, capacity)];. (It reads my_tail - 1, which is the index of the last valid item).

__device__ bool isEmpty() / __device__ int getSize()

These are approximate, "snapshot-in-time" functions. They read the volatile head and tail counters and return (head >= tail) or (tail - head), respectively. They are inherently racy (in the sense that the state may have already changed by the time the value is returned), but they do not use locks and are safe to call.

3. **Block-level shared memory deque:**

The source code for this version is contained within the *src/3_blk_level_shared_deque* directory as a header file that can be included in client CUDA programs.
The main objective for this third design is latency and locality. The previous two implementations (lock-based and lock-free) resided in Global Memory. While this allows communication between any thread in the grid, Global Memory is physically far from the execution cores (high latency) and is a shared resource for the entire GPU (limited bandwidth).

Many parallel algorithms, such as local work-stealing or block-level task queues, only require communication within a block. For these scenarios, accessing Global Memory is an overkill and a performance bottleneck.

This design implements a Lock-Free Block-Level Deque that lives entirely in Shared Memory. Shared Memory is an on-chip, user-managed cache that is orders of magnitude faster than Global Memory (similar to L1 cache latency). By restricting the scope to a single block, we can achieve extremely high throughput and low latency for intra-block coordination.

The architecture of this implementation differs significantly from the previous two because it does not interact with the host (CPU) for initialization or memory management.

i. **No Host Handle:** Since Shared Memory exists only during the lifetime of a kernel block execution, there is no GpuDequeHandle. The deque is instantiated directly inside the kernel using the __shared__ keyword.
ii. **Template Capacity:** The capacity is defined as a template parameter CAPACITY (e.g., BlockDeque<int, 256>). This allows the compiler to statically

allocate the necessary shared memory at compile time, avoiding dynamic allocation overheads.

iii. **Scoped Atomics:** A crucial optimization in this design is the use of scoped atomics (atomicAdd_block, atomicSub_block). Standard atomics (atomicAdd) enforce coherence across the entire GPU (L2 cache and Global Memory). Scoped atomics inform the hardware that synchronization is only needed within the thread block. This bypasses the L2 cache entirely for atomic operations, resulting in significantly faster execution.

The core logic mirrors the "Claim-Check-Rollback" strategy of the global lock-free deque but is adapted for the specific constraints of shared memory.

i. **Initialization:** Unlike global memory which can be initialized by the host cudaMemcpy, shared memory is uninitialized garbage when a block starts. Therefore, an explicit init() function is provided. This must be called by exactly one thread (usually threadIdx.x == 0) at the beginning of the kernel, followed by a __syncthreads() barrier to ensure the deque is ready before other threads use it.

ii. **Unbounded Counters:** It uses the same unbounded head and tail counter strategy as the global lock-free version to manage concurrency without locks.

iii. **Buffer Storage:** The T buffer[CAPACITY] is a standard array embedded directly within the class structure. Because the class instance itself is placed in shared memory, this array is automatically in shared memory.

The following __device__ functions are exposed as interfaces to the deque. They are optimized for low-latency, intra-block usage.

__device__ void init()

- **Purpose:** Resets the head and tail counters to 0.
- **Usage Requirement:** Must be called by a single thread (e.g., if (threadIdx.x == 0) deque.init();) followed by __syncthreads() to ensure visibility to the rest of the block.

__device__ bool pushBack(const T& element)

- **Arguments:** const T& element - Data to push.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    1. **Claim:** int my_tail = atomicAdd_block(&tail, 1); uses the block-scoped atomic to claim a slot.
    2. **Check:** Checks if full: my_tail - (*(volatile int*)&head) >= CAPACITY.
    3. **Rollback:** If full, atomicSub_block(&tail, 1) restores the counter and returns false.
    4. **Commit:** Writes to buffer[positive_mod(my_tail, CAPACITY)]. It must be noted that memory fences are implicitly handled by the scoped semantics within the block.

__device__ bool popFront(T* out_element)

- **Arguments:** T* out_element - Pointer to store result.
- **Return Value:** bool - true on success, false if the deque is empty.
- **Logic:**
    1. **Claim:** int my_head = atomicAdd_block(&head, 1); claims a "pop" ticket.
    2. **Check:** Checks if empty: my_head >= (*(volatile int*)&tail).
    3. **Rollback:** If empty, atomicSub_block(&head, 1) restores counter and returns false.
    4. **Commit:** Reads from buffer[positive_mod(my_head, CAPACITY)] and returns true.

__device__ bool pushFront(const T& element)

- **Arguments:** const T& element - Data to push.
- **Return Value:** bool - true on success, false if the deque is full.
- **Logic:**
    1. **Claim:** int my_head = atomicSub_block(&head, 1); decrements head to claim a slot at the front.
    2. **Check:** Checks if full: (*(volatile int*)&tail) - my_head >= CAPACITY.
    3. **Rollback:** If full, atomicAdd_block(&head, 1) restores counter and returns false.
    4. **Commit:** Writes element to buffer[positive_mod(my_head - 1, CAPACITY)].

__device__ bool popBack(T* out_element)

- **Arguments:** T* out_element - Pointer to store result.
- **Return Value:** bool - true on success, false if the deque is empty.
- **Logic:**
    1. **Claim:** int my_tail = atomicSub_block(&tail, 1); decrements tail to claim the last item.
    2. **Check:** Checks if empty: (*(volatile int*)&head) >= my_tail.
    3. **Rollback:** If empty, atomicAdd_block(&tail, 1) restores counter and returns false.
    4. **Commit:** Reads from buffer[positive_mod(my_tail - 1, CAPACITY)].

__device__ bool isEmpty() / __device__ int getSize()

- **Logic:** Reads the volatile head and tail counters to provide an instantaneous snapshot of the deque's state (emptiness or size respectively).

Experimental Setup

All the experiments were performed and the results were gathered on the ***cuda5 CIMS*** machine which has the following GPU configuration:

Model	NVIDIA GeForce RTX 4070
Architecture	Ada Lovelace
Compute Capability	8.9
CUDA Version	13.0
Driver Version	580.76.05
Total GPUs	1
Number of SMs	46
Number of SPs per SM	128
Total SPs	5888 (46 * 128)
PCIe Version	Gen3 (Current), Gen4 (Device Max)
GPU Clock	1,920 MHz (Current)
Max GPU Clock	3,105 MHz
Power Draw	96.6 W
Power Limit	200 W

The *nvcc* CUDA compiler version 13.0 was used to compile and build all the implemented CUDA programs.

Details regarding the *src* folder hierarchy, *benchmarks* folder hierarchy, steps and commands to execute the different versions and the benchmarks are detailed in the attached ***readme.txt*** file.

To evaluate the proposed GpuDeque library, a comprehensive testing suite was developed. The experiments are divided into two categories: **Functionality Verification** (to ensure logical correctness and type safety) and **Performance Benchmarking** (to measure execution time and compare the different implementation versions).

Functionality Verification

It is critical to verify that the implemented data structure behaves correctly according to the fundamental definition of a double-ended queue. Furthermore, since the library is designed as a C++ template supporting generics, it must be proven to handle various data types—from simple primitives to complex custom structures—without memory corruption or alignment issues.

This test suite is housed in the *func_tests.cu* file under *benchmarks/functional*. It is designed to execute a single-threaded kernel (<<<1, 1>>>) to isolate the logical correctness of the circular buffer and pointer arithmetic from the complexities of concurrency.

The core verification kernel, *verify_logic_kernel*, performs a specific sequence of operations designed to test both ends of the deque simultaneously. It uses three distinct values (A, B, C) to construct a specific state and then deconstructs it to verify the retrieval order.

The sequence is as follows:

1. **Push Back (A):** The deque contains [A].

2. **Push Front (B):** The deque contains [B, A].
3. **Push Back (C):** The deque contains [B, A, C].

Once the deque is in the state [B, A, C], the test verifies the retrieval logic:

1. **Pop Front:** Must return B. (Remaining: [A, C])
2. **Pop Back:** Must return C. (Remaining: [A])
3. **Pop Front:** Must return A. (Remaining: Empty)

This sequence confirms that the front and rear pointers (and their wrap-around logic) are functioning correctly relative to each other.

A robust GPU library must handle more than just integers. To validate the templated design, the test defines two custom structures that mimic real-world HPC workloads:

- **DeviceComplex:** A structure representing complex numbers with float r (real) and float i (imaginary) components. This tests the deque's ability to handle multi-field structures and ensures that the compiler correctly calculates sizeof(T) for memory allocation and pointer arithmetic. Custom operator== and operator!= overloads are implemented to handle floating-point epsilon comparisons on the device.
- **FixedString<N>:** Unlike CPU C++, CUDA does not support dynamic std::string. To test text processing capabilities, a FixedString template is created. This structure holds a fixed-size character array (e.g., char data[8]). Testing with this type validates that the deque correctly performs deep copies of array data when elements are pushed and popped, rather than just copying pointers.

The test harness runs the main logic test four times, instantiated with different types:

1. **int:** Basic primitive test.
2. **float:** Floating point arithmetic test.
3. **DeviceComplex:** Structured data test.
4. **FixedString<8>:** Array-based object test.

The kernel uses an atomicAdd on a unified error counter (int* errors) residing in global memory. If any value retrieved during the Pop phase does not match the expected value, the kernel prints a detailed error message (Expected vs. Got) and increments the error counter. The host checks this counter after synchronization; a value of 0 indicates a **PASS**, while any other value indicates a **FAIL**.

Apart from the above functionality tests, the performance benchmarks also test for a variety of scenarios for large N, in a concurrent environment.

The *benchmarks/functional* directory contains a Makefile which can be used to build the target executables. These target executables can then be executed to run the functionality tests for the different GPU Deque implementations. This is further detailed in the attached *readme.txt* file.

Performance Benchmarking

Global Deque benchmarks:

These benchmarks are housed under *benchmarks/performance/global_deque_benchmarks*. They are designed to verify and quantify the performance of the global deque versions implemented, namely, the lock-based and lock-free versions. These benchmarks are designed to saturate the GPU by launching a massive grid of threads (1,000,000 items processed), forcing significant contention on the atomic operations that manage the Deque's head and tail indices. A grid (391 blocks and 256 threads per block) consisting of 100,000 total threads is launched as part of these benchmarks where every thread processes (pushes or pops) 10 items.

FIFO_benchmark.cu: This evaluates the standard First-In-First-Out (Queue) usage pattern. This configuration tests the pushBack and popFront interfaces, which are the most common operations for work-stealing schedulers and producer-consumer pipelines.

- **Workload:** The benchmark attempts to push 1,000,000 unique integers (ids ranging from 0 to N-1) into the deque and then retrieve them.
- **Kernel 1 (push_back_kernel):** Each thread calculates a unique ID and attempts to insert it into the rear of the deque using pushBack. Atomic failures (due to capacity limits) are tracked.
- **Kernel 2 (pop_front_kernel):** This is executed after the push kernel completes, where threads attempt to remove items from the front using popFront. The retrieved values are stored in a global results array.
- **Metric:** The primary metric recorded is kernel execution time

LIFO_benchmark.cu: To ensure the library is symmetric and performant across all access points, the second benchmark tests the complementary operations: pushFront and popBack. While it is labelled as "LIFO", logically, this setup acts as a reverse-direction queue. This is crucial for verifying that the atomic contention logic works equally well at both ends of the circular buffer and that no bias exists in the implementation of the head vs. the tail counters.

- **Kernel 1 (push_front_kernel):** Threads contend to insert unique integers into the *front* of the deque.
- **Kernel 2 (pop_back_kernel):** Threads contend to remove items from the *rear* of the deque.
- The workload and metrics are similar to the FIFO_benchmark

Benchmarking concurrent data structures is risky if data integrity is not strictly monitored. To verify correctness without the massive overhead of sorting the output array on the host, a mathematical summation checksum is utilized. Since we push a known sequence of integers (0 to N−1), the sum of the popped elements must equal the sum of an arithmetic progression: $Sum = N \times (N+1)/2$

Here, Sum = (N-1)x(N-1+1)/2 = Nx(N-1)/2

The *verify_results* function performs three checks:

1. **Summation Check:** Calculates the actual sum of popped items and compares it to the expected formula.
2. **Push/Pop Failure Check:** Ensures that for a capacity N and input N, exactly N pushes and N pops succeeded.
3. **Uniqueness Check:** Uses a boolean vector to ensure no duplicates were retrieved, confirming that the locks/atomic counters correctly prevented race conditions.

Global Deque benchmarks for the stdgpu library:

To rigorously evaluate the performance of our custom GpuDeque implementations, it is essential to compare them against an established, open-source standard. For this purpose, the stdgpu open source C++ library was chosen. These benchmarks are housed under *benchmarks/performance/stdgpu_benchmarks.*

While the workload topology (number of threads, items, and blocks) remains identical to the benchmarks described in the previous section, the stdgpu API differs slightly from our custom interface. Consequently, specific adapter kernels were developed to ensure an "apples-to-apples" comparison.
The key implementation differences handled in the benchmark include:

1. **Device Object Creation:** Unlike our pointer-based handle, stdgpu requires a specific creation routine (stdgpu::deque<int>::createDeviceObject(CAPACITY)) to instantiate the container and its internal memory manager on the device.
2. **Return Semantics:** Our implementation uses a boolean return with a pointer argument (bool pop(T* out)). In contrast, stdgpu typically returns a pair or a value-wrapper containing both the data and a validity flag. The benchmark kernels were adapted to unpack this return value (ret.first for data, ret.second for success) to maintain the same logic flow.

FIFO_benchmark.cu: This test replicates the high-contention queue workload used in the previous section but utilizes the stdgpu container.

- **Push Phase:** The push_back_kernel utilizes d_deque.push_back(item_id). It handles atomic contention internally within the library.
- **Pop Phase:** The pop_front_kernel utilizes d_deque.pop_front().
- **Verification:** The same summation and uniqueness verification logic is applied. This ensures that stdgpu correctly handles the massive concurrency of 1,000,000 threads without data loss.

LIFO_benchmark.cu: This test replicates the "reverse" usage pattern to ensure the stdgpu library is also symmetric in its performance characteristics.

- **Push Phase:** Threads contend to push to the front using d_deque.push_front(item_id).
- **Pop Phase:** Threads contend to pop from the back using d_deque.pop_back().
- **Verification:** The same summation and uniqueness verification logic is applied.

The stdgpu library is downloaded/cloned from the GitHub Repository (https://github.com/stotko/stdgpu). This is then built to generate the required library files

which are used for executing the benchmarks. Further details regarding the method to clone, build and generate the library files for stdgpu are mentioned in the attached *readme.txt* file.

Block Deque benchmarks:

These benchmarks are housed under *benchmarks/performance/block_deque_benchmarks*. While the previous benchmarks measured grid-wide contention on a single deque, many parallel algorithms (such as localized task queues or hierarchical work-stealing) require independent deques for each thread block. To evaluate this use case, a Producer-Consumer benchmark that instantiates a unique deque for every thread block in the grid was developed. This setup compares the performance of the Global Memory implementations (Version 1 & 2) against the specialized Shared Memory implementation (Version 3).

The test configuration simulates a high-density workload where threads within a block must exchange data rapidly:

- **Grid Configuration:** 1024 Blocks, 512 Threads per block.
- **Role Division:** The block is split evenly. Threads 0-255 act as **Producers**, and threads 256-511 act as **Consumers**.
- **Workload:** Each producer pushes 5 items. A __syncthreads() barrier separates the phases. Consumers then consume the data until the deque is empty. This results in approximately 1.3 million total operations across the grid.

The benchmarks for the global deques and block level deques although similar in configuration and core logic slightly differ in their structure as follows:

- **Global Deque Array Test** (*glb_deque_FIFO.cu* and *glb_deque_LIFO.cu*): This benchmark tests the overhead of using Global Memory for block-local communication.
    - **Setup:** The host allocates an array of pointers (GpuDeque<int>**), where each pointer corresponds to a distinct deque instance allocated in Global Memory.
    - **Kernel Logic:** Each block reads its assigned deque pointer based on blockIdx.x. Although the contention is reduced (limited to 512 threads per deque rather than the whole grid), the atomic operations must still traverse the memory hierarchy to L2 or global DRAM, incurring higher latency.
- **Block-Level Shared Deque Test** (*blk_deque_FIFO.cu* and *blk_deque_LIFO.cu*): This benchmark evaluates the specialized BlockDeque implementation.
    - **Setup:** No external memory is allocated. The deque is instantiated dynamically using __shared__ BlockDeque<int, CAPACITY>.
    - **Kernel Logic:** Thread 0 initializes the deque. The producers and consumers then interact using **scoped atomics** (atomicAdd_block).
    - **Significance:** This test is designed to demonstrate the massive latency reduction achieved by keeping data on-chip. By bypassing Global Memory and the L2 cache, and using shared memory atomics, this implementation is expected to outperform the global versions in this specific topology.

Similar to the grid-wide tests, both the Global Array and Shared Memory benchmarks are run in two modes:

1. **FIFO:** Producers use pushBack, Consumers use popFront.
2. **LIFO:** Producers use pushFront, Consumers use popBack.

This ensures that the block-level logic maintains symmetry and correctness regardless of the queueing strategy used. Verification is performed by marking a global results array; if all items are successfully retrieved (marked as 1), the test passes.

Each of the 3 categories of benchmarks housed under *benchmarks/performance* contain a Makefile which can be used to build all the target executables. These target executables can then be executed to run the appropriate benchmark suite. This is further detailed in the attached *readme.txt* file.

Results and Analysis

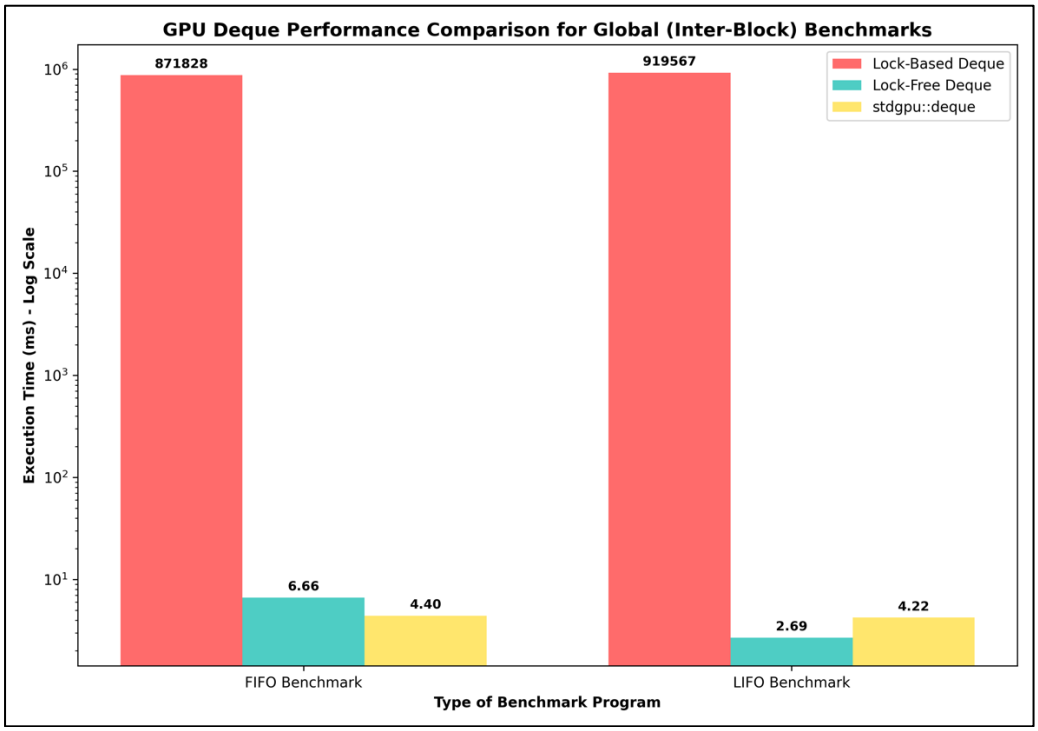
Functionality Verification:

- It was verified that the functionality tests described in the previous section passed successfully for the Deque Implementations, specifically the lock based and the lock free global deque implementations.
- Further, all of the performance benchmarks also include extensive tests in a concurrent environment that verify the behaviour of the deque under different scenarios. It was verified that all these pass successfully for all the different versions implemented, including the lock based and lock free global deque, the block level shared memory deque and the deque exposed by the stdgpu library.
- The particular functional tests and verification methodology have been detailed under appropriate subsections of the previous Experimental Setup section.

Performance Benchmarking:

Global Deque Benchmarks

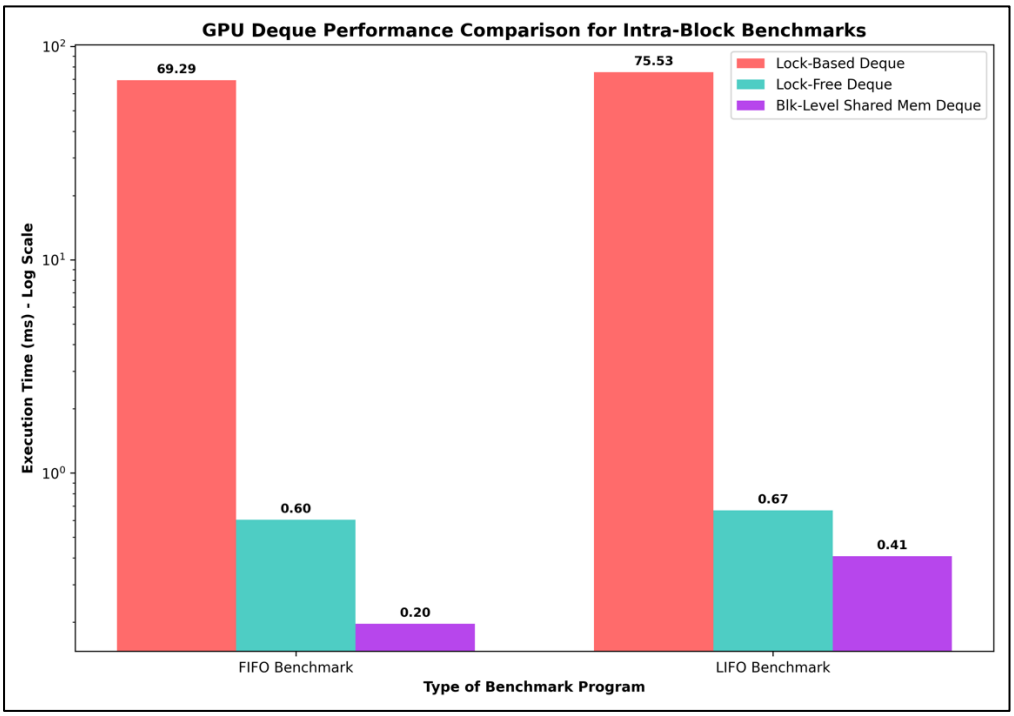
The below graph summarizes the generated results for the Global (Inter-Block) Deque Benchmarks described in the Experimental Setup section. These results correspond to the 2 versions implemented as part of this project, namely, Lock-based Deque and Lock-Free Deque. It also includes the results generated for deque exposed by the stdgpu library. The Execution Time (in milliseconds) of the kernels corresponding to the benchmarks is used as the metric for comparison. (All the benchmark results are generated by averaging over multiple runs)



The Analysis of the results from the above plot are as follows:

- It can be clearly seen that the Lock-Based Deque has the worst performance among all Deque implementations, for both the FIFO and LIFO benchmark suites.
- This is completely justified due to the use of an explicit global spin lock (through atomic CAS) in the Lock-Based Deque. This global lock on the deque completely serializes all the operations and hence all threads except one are blocked at any point in time. This severely impacts the performance in a negative manner and is corroborated by the large execution time for both the benchmarks.
- The Lock-Free Deque and stdgpu::deque show a considerable improvement in performance with more than 100000x Speedup as compared to the Lock-Based version. This is mainly because these versions avoid the use of expensive global locks and utilize other cheaper alternatives like atomic counters to implement a “ticketing” strategy for the slots of the deque in the lock free version.
- It must also be noted that the lock-free deque version implemented as part of this project has similar orders of performance to that of the open source stdgpu deque. The lock-free version experiences an approximate 50% slowdown as compared to the stdgpu deque for the FIFO benchmark. However, the lock-free version experiences a 36% improvement as compared to the stdgpu deque for the LIFO benchmark.
- The reasons for the difference in performance between lock-free version and the stdgpu version could be speculated as follows: As stdgpu is a production grade library, it likely implements a chunked memory layout rather than a single contiguous array. It could use internal optimizations for memory coalescing and efficient cache usage. It could be possible that some of these optimizations are proving to be beneficial in the case of the FIFO benchmark but are detrimental in the case of the LIFO benchmark. These reasons are purely speculative and need to be investigated further.

Block Deque benchmarks:
The below graph summarizes the generated results for the Block (Intra-Block) Deque Benchmarks described in the Experimental Setup section. These results correspond to the 3 versions implemented as part of this project, namely, Lock-based Deque, Lock-Free Deque, and the Block-level shared memory Deque. The Execution Time (in milliseconds) of the kernels corresponding to the benchmarks is used as the metric for comparison. (All the benchmark results are generated by averaging over multiple runs)



The Analysis of the results from the above plot are as follows:

- It can be clearly seen that the Lock-Based Deque has the worst performance among all Deque implementations, for both the FIFO and LIFO benchmark suites.
- This is completely justified due to the use of an explicit global spin lock (through atomic CAS) in the Lock-Based Deque. This global lock on the deque completely serializes all the operations and hence all threads except one are blocked at any point in time. This severely impacts the performance in a negative manner and is corroborated by the large execution time for both the benchmarks.
- The Lock-Free Deque and the Block-level Shared Memory Deque show a considerable improvement in performance, with more than 100x Speedup as compared to the Lock-Based Deque. This is mainly because these versions avoid the use of expensive global locks and utilize other cheaper alternatives like atomic counters to implement a “ticketing” strategy for the slots of the deque in the lock free version.
- It can also be seen that the Block-level Shared Memory Deque has the best performance and demonstrates an improvement of 66.67% for the FIFO benchmark and 38.8% for the LIFO benchmark as compared to the Lock-free Deque. This is despite the fact that both these versions implement the deque in a similar lock-free manner using atomic counters on the buffer slots. The primary reason for the performance improvement is the manner in which the memory buffer associated

with the Deque is structured. The Lock-free Deque houses its buffer in the Global Memory whereas the Block-level Deque houses its buffer in the Shared Memory. It is known that Global Memory incurs a higher Latency for memory access as compared to the faster and nearer Shared Memory. As these benchmarks specifically test for the scenarios where a deque is used for Intra-Block coordination, the Shared memory version outperforms the Global memory version.

Conclusion

- This project implements different versions of the Deque Data Structure and a library of its common operations for GPU Programming. This is provided as an easy to integrate, header-only library. It implements 2 versions of the Global Deque for use across different blocks of a grid. These include a naïve global lock-based implementation which serializes all its operations and a lock-free version that avoids the use of global locks by implementing "ticketing" for the slots of the buffer using atomic counters. Further, a 3rd implementation named Block-level Deques for Intra-Block communication has been developed. It uses ideas similar to the lock-free version but contains the entire deque within shared memory, hence, being better suited for applications that only require a deque within the context of a block.

- This project also implements various benchmarks to verify the functionality and compare the performance of the different versions implemented. The Functionality benchmark tests the core logic of the deque and its ability to be generic to support all data types. The performance benchmarks categorized as FIFO and LIFO contain more extensive scenarios in a concurrent environment. The project develops performance benchmarks for both the Global and the Block-level deques. In order to benchmark the performance against a production grade library, the stdgpu::deque is used and compared against the implemented versions for different benchmarks. The performance results of the lock-free version against the stdgpu::deque are quite promising, showing a 36% improvement for the LIFO benchmark. Further, the block-level shared memory deque demonstrates a significant improvement, of up to 67% as compared to the lock free version, highlighting its suitability for applications requiring a deque only for Intra-Block coordination.

- Admittedly, there is substantial scope for future work. Despite the positive performance improvement for the LIFO benchmark, the lock-free version experiences a 50% slowdown as compared to the stdgpu::deque for the FIFO benchmark. This needs to be investigated further to ascertain possible reasons. Consequently, relevant optimizations like memory coalescing, chunking, and improving cache efficiency can be implemented to augment the deque. Further, one of the drawbacks of the current design is that it uses a fixed size buffer and does not allow for resizing the deque. It needs to be explored further to deduce if resizing the deque can be supported while addressing all the possible issues like no possibility to invoke CudaMalloc from within a kernel to allocate a resized buffer, no global synchronization to stop all threads across the grid while resizing, and no guarantee regarding pointer coherency, that is if all the threads will see the new address atomically.

References

- David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, in Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

- Nathan Bell and Jared Hoberock, Thrust: A Productivity-Oriented Library for CUDA, in GPU Computing Gems Jade Edition, Morgan Kaufmann, October 2011.

- Patrick Stotko, stdgpu: Efficient STL-like Data Structures on the GPU, in arXiv:1908.05936, August 2019.

- Patrick Stotko, Stefan Krumpen, Matthias B. Hullin, Michael Weinmann, and Reinhard Klein, SLAMCast: Large-Scale, Real-Time 3D Reconstruction and Streaming for Immersive Multi-Client Live Telepresence, in IEEE Transactions on Visualization and Computer Graphics, Vol. 25, No. 5, pp. 2102–2112, May 2019.

- Duane Merrill and Michael Garland, CUB: CUDA Unbound, in NVIDIA Research, 2016.

- Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton, Thread Scheduling for Multiprogrammed Multiprocessors, in Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), June 1998.

- Jeff A. Stuart and John D. Owens, Message Passing on Data-Parallel Architectures, in 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), May 2011.

- Stanley Tzeng, Anjul Patney, and John D. Owens, Task Management for Irregular Parallel Workloads on the GPU, in Proceedings of High Performance Graphics (HPG), June 2010.

- David Chase and Yossi Lev, Dynamic Circular Work-Stealing Deque, in Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), July 2005.

- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, Cilk: An Efficient Multithreaded Runtime System, in Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), July 1995.

- Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala, ParlayLib: A Toolkit for Parallel Algorithms on Shared-Memory Multicore Systems, in Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), July 2020.