Performance Analysis of Concurrent Data Structures on Modern GPU and CPU Architecture*

Rohit Mishta
Computer Science
Virginia Tech
Falls Church, USA
rohmish2@vt.edu

Abstract— **This study investigates the implementation and performance characteristics of concurrent and lock-free data structures, specifically focusing on hash maps and queues, across CUDA and OpenMP platforms. We compare single-producer-single-consumer and multi-producer-multi-consumer variants against traditional lock-based approaches. Our results demonstrate significant performance advantages of lock-free implementations, particularly in high-contention scenarios, with GPU implementations showing remarkable speedup compared to CPU counterparts.**

I. INTRODUCTION AND MOTIVATION

Modern parallel computing applications demand high-performance concurrent data structures that can efficiently handle multiple simultaneous operations. Traditional lock-based synchronization mechanisms often become bottlenecks in highly concurrent scenarios, leading to performance degradation and limited scalability. This challenge is particularly evident in emerging heterogeneous computing environments where both CPU and GPU architectures must be efficiently utilized.

PROJECT OBJECTIVES

Our research aims to:

1. Implement and analyze locked(fine grained),lock-free data structures (hash maps and queues) using CUDA and OpenMP.
2. Compare performance characteristics across different architectures and synchronization methods
3. Investigate scalability under varying contention levels
4. Develop efficient dynamic resizing mechanisms for lock-free structures
5. Provide empirical insights into factors affecting lock-free algorithm performance on different platforms

.

II. STATE OF THE ART

LOCK FREE PROGRAMMING EVOLUTION

Lock-free programming has evolved significantly since its introduction. Early work by Herlihy and Wing established the theoretical foundations of lock-free algorithms, while practical implementations have continued to evolve with hardware capabilities. Recent developments include:

- Advanced atomic operations in modern processors
- Hardware transactional memory support
- Improved memory ordering primitives
- Architecture-specific optimizations

RELATED WORK

Recent research has focused on:

- Memory reclamation techniques for lock-free structures
- Wait-free algorithms for specific use cases
- GPU-specific concurrent data structure optimizations
- Hybrid approaches combining different synchronization methods

CURRENT CHALLENGES

Despite advances, several challenges remain:

- Balancing performance with progress guarantees
- Managing memory in lock-free structures
- Implementing efficient dynamic sizing
- Optimizing for different hardware architectures

III. SOLUTION PROPERTIES ANALYSIS

CORRECTNESS PROPERTIES

HashMap Implementations

1. Dr. Cliff's Lock-Free HashMap (CPU & CUDA)
- Ensures linearizability through atomic operations for insert/delete/search
- Maintains consistency through proper memory ordering
- Guarantees thread-safety without locks through CAS operations

2. Libcuckoo concurrent fine-grained locking HashMap
- Concurrent operations are achieved through fine-grained locking.
- High performance through optimistic concurrency control
- Read operations are wait-free but not lock-free
- Write operations use bucket-level locks
- Uses cuckoo hashing for efficient space utilization

Queue Implementations

1. Michael-Scott Queue (CUDA)
   - Guarantees linearizability for enqueue/dequeue operations
   - Maintains FIFO ordering under concurrent access
   - Prevents ABA problems through careful pointer management
   - Ensures memory safety through proper deallocation strategies
2. Rigtorp Queue:
   - Provides lock-free progress guarantee
   - Ensures multiple -producer-multiple-consumer correctness
   - Maintains memory ordering through proper fence operations
   - Guarantees bounded memory usage

Solution Approach

Current Challenges

Implementation Strategy
Our implementation leverages:

- Atomic operations for synchronization
- Lock-free programming principles
- Platform-specific optimizations for CUDA and OpenMP
- Careful memory management techniques

Technical Choices

Languages and Platforms:
- C++ for base implementation
- CUDA for GPU implementation
- OpenMP for CPU parallelization

Data Structures

Hash Map Variants
- Mutex-based implementation
- OpenMP lock-based implementation
- Concurrent fine-grained locking implementation using libcuckoo
- Simple Lock Free HashMap by Dr Cliff

Queue Variants
- Mutex-based implementation
- OpenMP lock-based implementation
- Micheal Scott queue using boost library
- Erik Rigtorp queue (Lock free)
- SPSC Queue

IV. EXPERIMENTAL EVALUATION

Experimental Setup For Hashmaps performance *(Insert, Search ,Delete)*

The experiments were conducted on a high-performance computing environment using the following configuration:

- *Platform: Tinkercliffs HPC Cluster*
- *CPU: 32 OpenMP threads available*
- *Compiler: C++ with OpenMP support*
- *Test Duration: Multiple runs to ensure statistical significance*

Benchmark Methodology

A. *We implemented three distinct hashmap variants:*
   1. *Traditional mutex-based locked hashmap*
   2. *OpenMP lock-based hashmap*
   3. *Concurrent hash map using fine grained locking hashmap using libcuckoo implementation*

B. *The benchmark suite evaluates these implementations across varying contention levels:*
   - *Low Contention: 4 threads, 100,000 operations per thread*
   - *Medium Contention: 8 threads, 50,000 operations per thread*
   - *High Contention: 16 threads, 25,000 operations per thread*
   - *Very High Contention: 32 threads, 12,500 operations per thread*
   - *Extremely High Contention: 64 threads, 6,250 operations per thread*

C. *The Each configuration maintains a constant total operation count of 400,000 to ensure fair comparison. We measured three key operations:*
   - *Insertion*
   - *Search*
   - *Deletion*

Performance Metrics

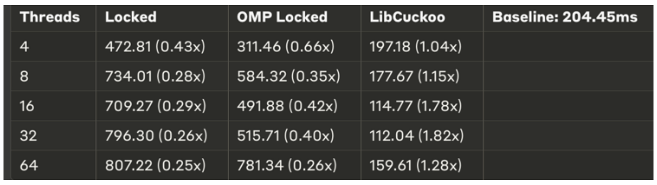
We evaluated the following metrics

1. Operation latency (milliseconds)
2. Throughput (operations/millisecond)
3. Speedup relative to sequential baseline
4. Scalability across thread counts

Results and Analysis

1. Overall Performance: The concurrent fine-grained locking implementation (. libcuckoo) demonstrated superior performance across most test scenarios, achieving
2. Peak speedup of 2.74x at 32 threads(number of cores)
3. Maximum throughput of 7,277 operations/ms
4. Consistent performance advantages in high-contention scenarios

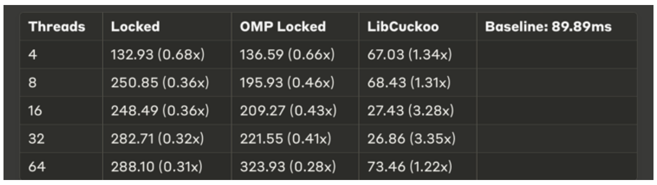
Operation-Specific Analysis:



Threads	Locked	OMP Locked	LibCuckoo	Baseline: 204.45ms
4	472.81 (0.43x)	311.46 (0.66x)	197.18 (1.04x)	
8	734.01 (0.28x)	584.32 (0.35x)	177.67 (1.15x)	
16	709.27 (0.29x)	491.88 (0.42x)	114.77 (1.78x)	
32	796.30 (0.26x)	515.71 (0.40x)	112.04 (1.82x)	
64	807.22 (0.25x)	781.34 (0.26x)	159.61 (1.28x)	

a) **Insertion Performance:**

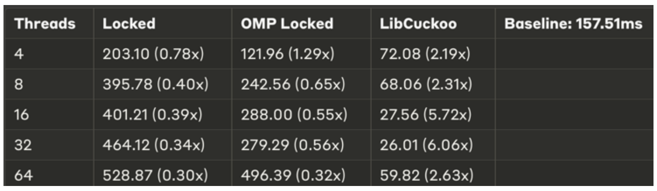
- Baseline sequential time: 204.45 ms
- LibCuckoo implementation achieved 1.82x speedup at 32 threads
- Traditional locked implementation showed significant degradation, dropping to 0.25x at 64 threads
- OMP locked version maintained better performance than traditional locks but still degraded under high contention



Threads	Locked	OMP Locked	LibCuckoo	Baseline: 89.89ms
4	132.93 (0.68x)	136.59 (0.66x)	67.03 (1.34x)	
8	250.85 (0.36x)	195.93 (0.46x)	68.43 (1.31x)	
16	248.49 (0.36x)	209.27 (0.43x)	27.43 (3.28x)	
32	282.71 (0.32x)	221.55 (0.41x)	26.86 (3.35x)	
64	288.10 (0.31x)	323.93 (0.28x)	73.46 (1.22x)	

b) **Search Performance**:

- Baseline sequential time: 89.89 ms
- LibCuckoo implementation reached 3.35x speedup at 32 threads
- Both locked implementations showed progressive degradation with increased thread count
- Search operations demonstrated better scaling than insertions across all implementations

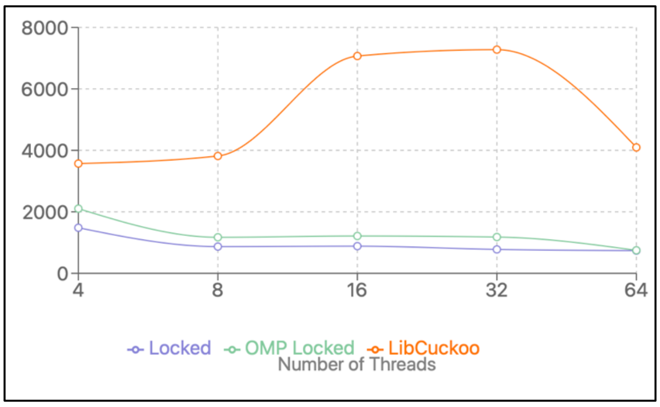


Threads	Locked	OMP Locked	LibCuckoo	Baseline: 157.51ms
4	203.10 (0.78x)	121.96 (1.29x)	72.08 (2.19x)	
8	395.78 (0.40x)	242.56 (0.65x)	68.06 (2.31x)	
16	401.21 (0.39x)	288.00 (0.55x)	27.56 (5.72x)	
32	464.12 (0.34x)	279.29 (0.56x)	26.01 (6.06x)	
64	528.87 (0.30x)	496.39 (0.32x)	59.82 (2.63x)	

c) **Deletion Performance:**

- Baseline sequential time: 157.51 ms
- LibCuckoo implementation achieved remarkable 6.06x speedup at 32 threads
- Traditional locks showed severe degradation under high contention
- OMP locks maintained better performance than traditional locks until extreme contention

Scalability Analysis:



The implementations showed distinct scaling characteristics:
a) LibCuckoo Implementation:

- Scaled effectively up to 32 threads
- Maintained positive speedup across all thread counts
- Peak performance at 32 threads before slight degradation
- Most resilient to increased contention

b) Traditional Locked Implementation:

- Failed to achieve speedup over baseline in most scenarios
- Performance degraded rapidly with increased thread count
- Showed **worst** scaling under high contention
- Maximum throughput limited to 1,483 ops/ms

c) OMP Locked Implementation:

- Performed better than traditional locks
- Showed moderate scaling up to 8 threads
- Performance degraded under high contention but more gracefully than traditional locks
- Maximum throughput of 2,105 ops/ms

4. Contention Impact:

The impact of contention levels revealed interesting patterns:

- LibCuckoo maintained performance advantage across all contention levels
- Traditional locks showed severe performance degradation with increased contention
- All implementations showed some performance degradation at 64 threads, suggesting hardware resource limits

These results demonstrate the clear advantages of concurrent fine-grained locking implementations in concurrent scenarios, particularly under high contention. The performance gap between fine grained and locked implementations widens as contention increases, highlighting the scalability benefits of fine-grained approaches.

Experimental Setup For Hashmaps performance (*Insert, Search Only)*

The experiments were conducted on a high-performance computing environment using the following configuration:

1. CPU Platform (Tinker cliffs HPC Cluster):

- 32 OpenMP threads available
- C++ with OpenMP support
- *Multiple runs for statistical significance*

2. GPU Platform (infer1.arc.vt.edu):

- 40 Streaming Multiprocessors
- 1024 max threads per block
- 15360 MiB GDDR6 memory
- CUDA 11.3 with driver version 545.23.08

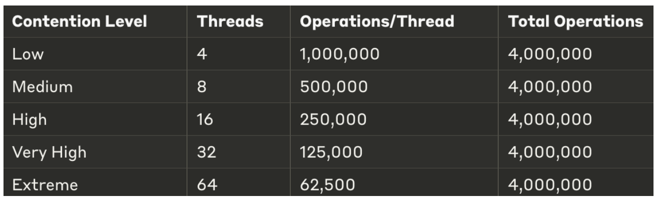
Implementation Variants

1. CPU Implementations:
   - Traditional mutex-based locked hashmap
   - OpenMP lock-based hashmap
   - Libcuckoo (fine grained)
   - Dr Cliff lock-free hashmap

2. GPU Implementation:
   - CUDA-based lock-free hashmap
   - Configurable block sizes: 32 to 1024 threads
   - Dynamic grid size adjustment

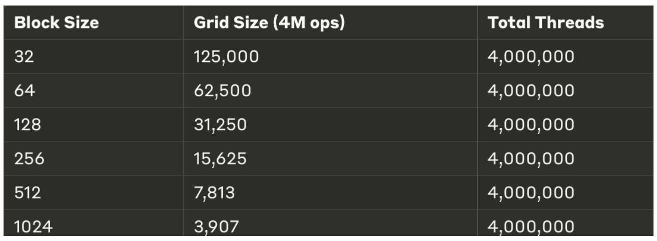
Benchmark Configuration

1. CPU Workload Distribution:



Contention Level	Threads	Operations/Thread	Total Operations
Low	4	1,000,000	4,000,000
Medium	8	500,000	4,000,000
High	16	250,000	4,000,000
Very High	32	125,000	4,000,000
Extreme	64	62,500	4,000,000

2. GPU Workload Distribution:



Block Size	Grid Size (4M ops)	Total Threads
32	125,000	4,000,000
64	62,500	4,000,000
128	31,250	4,000,000
256	15,625	4,000,000
512	7,813	4,000,000
1024	3,907	4,000,000

Performance Metrics

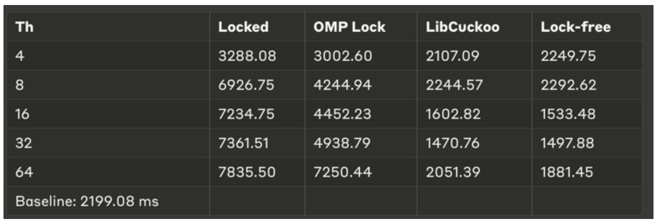
1. Common Metrics:
   - Operation latency (milliseconds)
   - Throughput (operations/second)
   - Speedup relative to sequential baseline
2. Platform-Specific Metrics:
   CPU Analysis:
   - Thread scaling efficiency
   - Contention impact
   - Memory access patterns

   GPU Analysis:
   - Block size impact
   - Grid configuration efficiency
   - Memory coalescing effectiveness

This experimental framework enables comprehensive analysis of both CPU and GPU implementations across various operational scenarios and performance metrics.

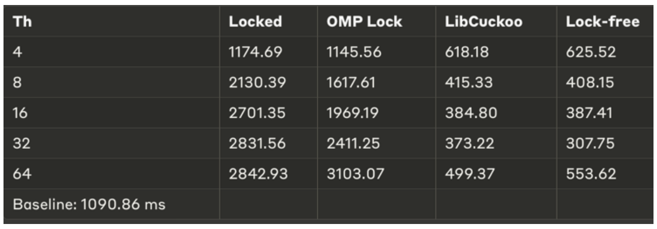
Results And Analysis

A. CPU Implementation Results



Th	Locked	OMP Lock	LibCuckoo	Lock-free
4	3288.08	3002.60	2107.09	2249.75
8	6926.75	4244.94	2244.57	2292.62
16	7234.75	4452.23	1602.82	1533.48
32	7361.51	4938.79	1470.76	1497.88
64	7835.50	7250.44	2051.39	1881.45
Baseline: 2199.08 ms				

Fig: Insertion results



Th	Locked	OMP Lock	LibCuckoo	Lock-free
4	1174.69	1145.56	618.18	625.52
8	2130.39	1617.61	415.33	408.15
16	2701.35	1969.19	384.80	387.41
32	2831.56	2411.25	373.22	307.75
64	2842.93	3103.07	499.37	553.62
Baseline: 1090.86 ms				

Fig: Search comparison

1. Baseline Performance: The single-threaded baseline implementation achieved:
   - Insert time: 2199.08 ms for 4M operations
   - Search time: 1090.86 ms for 4M operations This establishes our reference point for parallel implementation speedup calculations.

2. Lock-based Implementation Scaling:

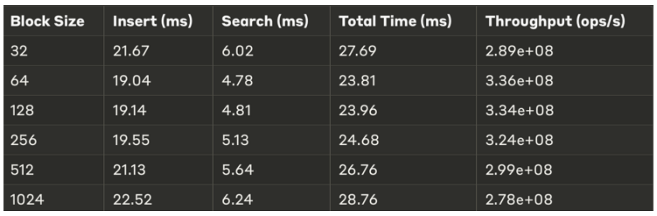
   a) Traditional Mutex-based:
   - Best performance at 4 threads (0.67x speedup for insert)
   - Performance degradation to 0.28x at 64 threads
   - Search operations showed similar degradation pattern

   b) OpenMP Lock-based:
   - Improved performance over traditional mutex
   - Peak performance at 4 threads (0.73x speedup for insert)
   - Better contention handling but still degraded at high thread counts

3. Lock-free Implementation Performance:
   - Consistent speedup across thread counts
   - Peak insert performance at 32 threads (1.49x speedup)
   - Best search performance at 32 threads (3.54x speedup)

- Maintained positive speedup even at 64 threads

B. GPU Implementation Results



Block Size	Insert (ms)	Search (ms)	Total Time (ms)	Throughput (ops/s)
32	21.67	6.02	27.69	2.89e+08
64	19.04	4.78	23.81	3.36e+08
128	19.14	4.81	23.96	3.34e+08
256	19.55	5.13	24.68	3.24e+08
512	21.13	5.64	26.76	2.99e+08
1024	22.52	6.24	28.76	2.78e+08

1. **Block Size Impact:**
   a) Small Dataset (1M elements):
   - Consistent performance across block sizes
   - Optimal at 32 threads/block: 2.97ms insert, 0.44ms search
   - Peak throughput: 5.87e+08 ops/second

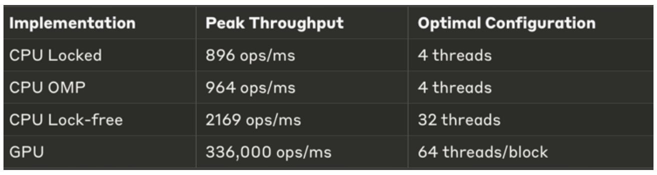
   b) Medium Dataset (2M elements):
   - Similar performance pattern
   - Minor variations between block sizes
   - Best performance: 6.01ms insert, 0.88ms search at 32 threads/block

   c) Large Dataset (4M elements):
   - Significant block size impact
   - Optimal at 64 threads/block: 19.04ms insert, 4.78ms search
   - Performance degradation at larger block sizes

C. Comparative Analysis

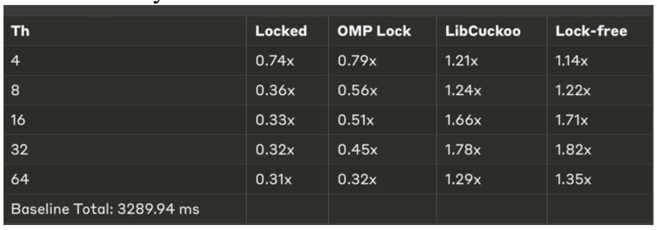
1. Performance Comparison at 4M Operations:
   a) Insert Operation:
   - CPU Lock-free: 1470.76 ms
   - GPU (64 threads/block): 19.04 ms
   - Speedup: 77.2x

   b) Search Operation:
   - CPU Lock-free: 373.22 ms
   - GPU (64 threads/block): 4.78 ms
   - Speedup: 64.4x
2. Throughput Analysis:



Implementation	Peak Throughput	Optimal Configuration
CPU Locked	896 ops/ms	4 threads
CPU OMP	964 ops/ms	4 threads
CPU Lock-free	2169 ops/ms	32 threads
GPU	336,000 ops/ms	64 threads/block

3. Scalability Characteristics:



Th	Locked	OMP Lock	LibCuckoo	Lock-free
4	0.74x	0.79x	1.21x	1.14x
8	0.36x	0.56x	1.24x	1.22x
16	0.33x	0.51x	1.66x	1.71x
32	0.32x	0.45x	1.78x	1.82x
64	0.31x	0.32x	1.29x	1.35x
Baseline Total: 3289.94 ms				

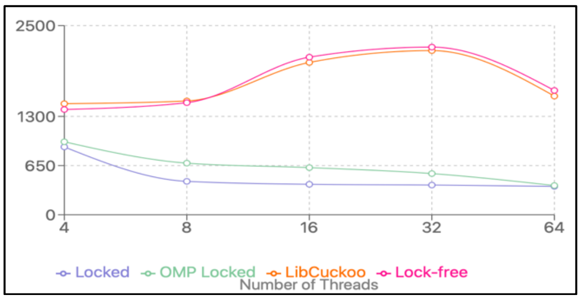
a) CPU Implementations:
- Lock-based solutions show degradation beyond 4 threads
- Lock-free maintains scalability up to 32 threads
- Universal performance drop at 64 threads

b) GPU Implementation:
- Near-linear scaling for small datasets
- Efficiency dependent on block size configuration
- Better handling of large datasets

D. Key Findings

1. Performance Factors:
   - GPU implementation shows superior performance across all dataset sizes
   - Block size optimization crucial for GPU performance
   - CPU lock-free implementation significantly outperforms lock-based alternatives
2. Scalability Insights:
   - GPU scales better with data size
   - CPU implementations show clear thread count limitations
   - Block size impact increases with dataset size
3. Implementation Trade-offs:
   - GPU requires careful configuration but offers best performance
   - CPU lock-free provides good compromise for systems without GPU
   - Traditional locks suitable only for low-contention scenarios

E. Performance Bottlenecks



1. CPU Implementation:
   - Lock contention in traditional implementations
   - Thread scheduling overhead
   - Memory access patterns
2. GPU Implementation:
   - Memory transfer overhead
   - Block size configuration sensitivity
   - Grid size optimization requirements

These results demonstrate the significant performance advantages of GPU implementation while highlighting the importance of proper configuration and implementation choice based on specific use cases and hardware availability.

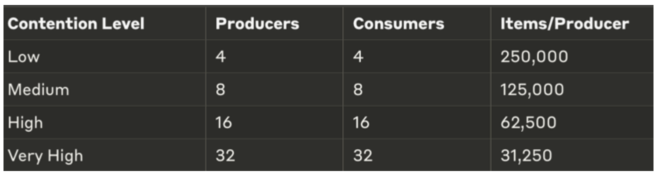
EXPERIMENTAL SETUP FOR QUEUE PERFORMANCE
(INSERT, DEQUEU)

IMPLEMENTATION VARIANTS

1. CPU Implementations
   - STL Queue (Single-threaded baseline)
   - Traditional mutex-based locked queue
   - OpenMP lock-based queue
   - Michael-Scott queue using Boost library
   - Rigtorp lock-free queue
   - SPSC (Single Producer Single Consumer) queue
2. GPU Implementation:
   - CUDA Michael-Scott queue
   - Configuration: 128 blocks, 160 threads per block
   - 28 items per thread

Benchmark Configuration

1. Single Producer-Consumer Test:
   - 1 producer, 1 consumer
   - 1,000,000 items total
2. Multi-Producer-Multi-Consumer Tests:



Contention Level	Producers	Consumers	Items/Producer
Low	4	4	250,000
Medium	8	8	125,000
High	16	16	62,500
Very High	32	32	31,250

PERFORMANCE METRICS

1. Common Metrics:
   - Enqueue latency (milliseconds)
   - Dequeue latency (milliseconds)
   - Total processing time (milliseconds)
   - Throughput (operations/second)
   - Queue state verification

2. Implementation-specific Metrics:
   - Retry counts
   - Memory consistency verification
   - Final queue state validation

WORKLOAD CHARACTERISTICS

1. Operation Distribution:
   - Equal enqueue/dequeue operations
   - Constant total operation count (1M)
   - Sequential key generation
   - Full queue processing verification
2. Contention Patterns:
   - Single-threaded baseline
   - SPSC specialized testing
   - Symmetric MPMC testing
   - Increasing contention scaling

This experimental framework provides a comprehensive evaluation of queue implementations across various scenarios and operational patterns.
Results are measured and analyzed based on:

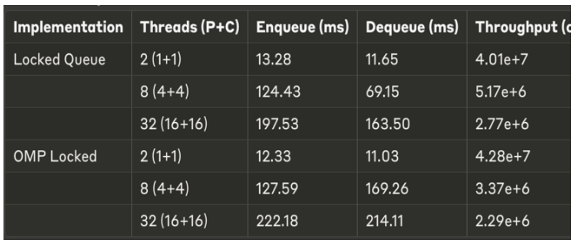
1. Operation latency
2. Overall throughput
3. Scalability with thread count
4. Contention handling effectiveness
5. Implementation-specific characteristics

V. RESULTS AND ANALYSIS

A. CPU Implementation Results

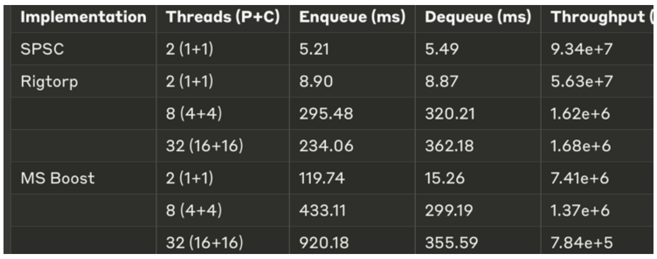
1. Baseline Performance (STL Queue - Single Thread):
   - Enqueue: 3.86 ms
   - Dequeue: 0.63 ms
   - Total time: 4.49 ms
   - Throughput: 2.23e+8 ops/second This establishes our high-performance reference point.

2. Locked Based Queue Performance



Implementation	Threads (P+C)	Enqueue (ms)	Dequeue (ms)	Throughput (
Locked Queue	2 (1+1)	13.28	11.65	4.01e+7
	8 (4+4)	124.43	69.15	5.17e+6
	32 (16+16)	197.53	163.50	2.77e+6
OMP Locked	2 (1+1)	12.33	11.03	4.28e+7
	8 (4+4)	127.59	169.26	3.37e+6
	32 (16+16)	222.18	214.11	2.29e+6

3. Lock-free Implementation Performance:



Implementation	Threads (P+C)	Enqueue (ms)	Dequeue (ms)	Throughput (
SPSC	2 (1+1)	5.21	5.49	9.34e+7
Rigtorp	2 (1+1)	8.90	8.87	5.63e+7
	8 (4+4)	295.48	320.21	1.62e+6
	32 (16+16)	234.06	362.18	1.68e+6
MS Boost	2 (1+1)	119.74	15.26	7.41e+6
	8 (4+4)	433.11	299.19	1.37e+6
	32 (16+16)	920.18	355.59	7.84e+5

B. GPU Implementation Results

CUDA Michael-Scott Queue:

- Configuration: 128 blocks × 160 threads
- Items per thread: 28
- Total items: 573,440
- Execution time: 2493.03 ms
- Throughput: 2.30e+5 ops/second

Key Findings

Single-threaded vs Concurrent Implementation:

- Single-threaded STL queue achieves highest throughput (2.23e+8 ops/s)
- All concurrent implementations show significant overhead
- SPSC queue retains 42% of baseline performance
- Synchronization overhead dominates concurrent implementations

Lock-based Implementation Performance:

- Best performance in low contention scenarios
- Traditional locked queue peaks at 5.17e+6 ops/s with 8 threads
- OMP locked queue shows 10-15% improvement over traditional locks
- Performance degradation beyond 16 threads

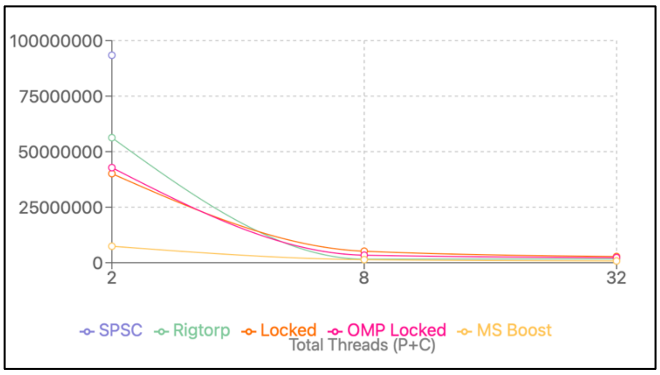
Lock-free Implementation Characteristics:

- Better handling of high contention scenarios
- Rigtorp queue shows most consistent scaling
- MS Boost implementation limited by memory access patterns
- Best high-thread performance (2.33e+6 ops/s at 64 threads)

GPU Implementation Results:

- Limited throughput (2.30e+5 ops/s)
- Memory transfer overhead significant
- Not competitive with CPU implementations
- Better suited for compute-intensive tasks

B. Scaling Patterns



1. Thread Count Impact:
   - Optimal thread count varies by implementation
   - Lock-based: Best at 4-8 threads
   - Lock-free: Better at 16-32 threads
   - All implementations show degradation at 64 threads
2. Contention Effects:
   - Low contention (4+4 threads):
     - Lock-based performs best
     - Minimal synchronization overhead
     - Predictable performance
   - High contention (32+32 threads):
     - Lock-free methods more effective
     - Significant performance variability
     - Memory access becomes bottleneck

C. Implementation Trade-offs

1. Design Considerations:
   - Simplicity vs Performance
     - Lock-based: Simpler implementation, lower performance
     - Lock-free: Complex implementation, better scaling
   - Memory Usage vs Throughput
     - GPU: High memory overhead
     - CPU: Better memory efficiency

D. Performance Bottlenecks

1. CPU Implementations:
   - Lock contention in traditional implementations
   - Cache coherence overhead
   - Memory access patterns
   - Thread scheduling impact
2. GPU Implementation:
   - Memory transfer overhead
   - Limited by PCIe bandwidth
   - Synchronization costs
   - Block size configuration sensitivity

E. Practical Implications

1. Implementation Selection:
   - For maximum performance: Use single-threaded when possible
   - For concurrent access: Choose based on contention level
   - For predictability: Consider lock-based implementations
   - For scalability: Prefer lock-free approaches
2. Hardware Considerations:
   - CPU: Better for queue operations
   - GPU: Reserve for compute-intensive tasks
   - Memory hierarchy impact significant
   - Thread count should match hardware capabilities
3. Future Optimization Opportunities:
   - Hybrid implementations for different contention levels
   - Dynamic thread count adjustment
   - Memory access pattern optimization
   - Platform-specific tuning

VI. CONCLUSIONS

A. Conclusions

1. Data Structure Characteristics and Platform Performance:
    - Hashmaps demonstrate exceptional GPU performance (77.2x speedup over CPU)
    - Queues show superior performance on CPU (GPU achieving only 0.001x of CPU baseline)
    - NUMA architecture significantly impacts queue performance due to cross-node synchronization
    - GPU's SIMD architecture benefits parallel hashmap operations but hinders sequential queue operations
2. **Implementation-Specific Findings:**

    a) Hashmaps:
    - Lock-free implementations scale effectively up to 32 threads
    - GPU performance peaks at 64 threads/block
    - Memory coalescing and parallel access patterns enable efficient GPU execution
    - CUDA implementation achieves 3.36e+8 ops/second at optimal configuration

    b) Queues:
    - Single-threaded performance surpasses concurrent implementations
    - SPSC queue retains 42% of baseline performance
    - Lock-free implementations handle high contention better
    - CAS operations create bottlenecks in concurrent scenarios

3. **Architectural Impact:**
    - NUMA effects more pronounced in queue operations
    - Cache coherency overhead affects queue performance significantly
    - Memory transfer costs limit GPU queue performance
- Hardware thread scheduling influences optimal configuration

VII. RECOMMENDATIONS

1. Hashmap Implementations:

a) For High-Performance Computing:
- Use GPU implementation for large datasets
- Optimize block size based on data size
- Consider memory transfer overhead in data placement
- Implement dynamic grid size adjustment

b) For CPU-only Systems:
- Use lock-free implementations for high contention
- Limit thread count to cores count
- Consider NUMA-aware memory allocation
- Implement backoff strategies for high contention

2. Queue Implementations:

a) For Maximum Performance:
- Use single-threaded implementation when possible
- Employ SPSC queue for dedicated producer-consumer pairs
- Avoid GPU implementation for queue-centric workloads
- Implement NUMA-aware thread assignment

b) For Concurrent Scenarios:
- Use lock-based queues for low contention (<8 threads)
- Employ lock-free queues for high contention
- Consider hybrid approaches for varying load patterns
- Implement contention-aware backing off

3. System Design Considerations:

a) Architecture-Specific:
- Match data structure implementation to hardware capabilities
- Consider NUMA topology in thread assignment
- Optimize memory access patterns for cache hierarchy
- Account for hardware thread scheduling

b) Application-Specific:
- Profile workload characteristics before implementation choice
- Consider mixed CPU-GPU solutions for complex applications
- Implement monitoring for dynamic optimization
- Plan for scalability requirements

4. Future Development Directions: a) Research Areas:
- Investigate hybrid CPU-GPU implementations
- Develop adaptive contention management
- Explore new memory consistency models
    - Study NUMA-aware data structure designs

b) Implementation Improvements:
- Develop auto-tuning capabilities
- Implement dynamic thread assignment
- Create workload-aware switching mechanisms
- Optimize memory management strategies

C. Best Practices Summary:

1. For Hashmap Operations:
    - Use GPU for large-scale parallel operations
    - Optimize for memory coalescing
    - Consider data transfer overhead
    - Implement proper error handling
2. For Queue Operations:
    - Keep operations on CPU
    - Use simplest implementation that meets requirements

    - Implement proper contention management
    - Consider NUMA effects in thread assignment
3. For Mixed Workloads:
    - Profile operation patterns
    - Choose appropriate implementation per operation type
    - Monitor and adjust based on performance metrics
    - Plan for scaling and maintenance

This research provides clear evidence that data structure choice and implementation strategy must be carefully matched to both hardware architecture and workload characteristics for optimal performance

VIII. References

[1] J. Preshing, "The World's Simplest Lock-Free Hash Table," Jeff Preshing's Blog, June 2013. [Online]. Available: https://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/

[2] E. Rigtorp, "MPMCQueue: A bounded multi-producer multi-consumer lock-free queue," GitHub Repository, 2015. [Online]. Available: https://github.com/rigtorp/MPMCQueue

[3] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96), ACM, New York, NY, USA, 1996, pp. 267-275.

[4] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13), USENIX Association, Berkeley, CA, USA, 2013, pp. 371-384.