High-Performance Multi-GPU Concurrent Queues: Case Study with Parallel Bellman-Ford SSSP

by

Beyza Çavuşoğlu

A Dissertation Submitted to the

Graduate School of Sciences and Engineering

in Partial Fulfillment of the Requirements for

the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

December 1, 2024

High-Performance Multi-GPU Concurrent Queues: Case Study with Parallel Bellman-Ford SSSP

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Beyza Çavuşoğlu

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

Assoc. Prof. Dr. Didem Unat (Advisor)

Prof. Yücel Yemez

Prof. Murat Manguoğlu

Date:

To my father, Bülent Çavuşoğlu, whose unmatched work ethic, determination, and integrity have been my lifelong inspiration, and to my mother, Sakine Çavuşoğlu, whose unwavering dedication and tireless support in my education shaped the foundation of my achievements—thank you for being my greatest guides and motivators. To my professor Didem Unat for her supports, to my lab mate, Mohammad Kefah Issa, for his generosity with his time, and to Muhammed Özdemir and Belgin Deryalar, whose constant motivation carried me through this journey—I owe them all a debt of gratitude beyond words.

ABSTRACT

High-Performance Multi-GPU Concurrent Queues: Case Study with Parallel Bellman-Ford SSSP
Beyza Çavuşoğlu
Master of Science in Computer Science and Engineering
December 1, 2024

This thesis presents a novel implementation of a concurrent FIFO queue algorithm for both single-GPU and multi-GPU environments, applied to the Single-Source Shortest Path (SSSP) Bellman-Ford algorithm. The primary contribution of this work is the design and implementation of a multi-GPU concurrent queue system using NVIDIA's NVSHMEM, which has not been previously explored. The Bellman-Ford algorithm is used as a case study to evaluate the performance of the proposed queue system, with this multi-GPU implementation being the first known instance of its kind. Experimental results demonstrate that the multi-GPU queue implementation achieves a maximum speedup of 3.92x and an average speedup of 3.04x over the single-GPU baseline on four NVIDIA A100 GPUs. When applied to the Bellman-Ford algorithm, the multi-GPU system achieves a maximum speedup of 3.794× and an average speedup of 3.573× compared to the single-GPU implementation, tested on a generated benchmark and 10 graphs of different kinds taken from the SuiteSparse Matrix Collection. These findings highlight the efficiency of the multi-GPU queue system for graph processing tasks and contribute to advancements in high-performance computing by addressing practical challenges in parallel computing.

ÖZETÇE

Yüksek Performanslı Çoklu-GPU Eşzamanlı Kuyruk Algoritmalarının Implementasyonu: Bellman-Ford SSSP'nin Paralelleştirilmesi
Beyza Çavuşoğlu
Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans
1 Aralık 2024

Bu tez, tek-GPU ve çoklu-GPU ortamlarında uygulanan yeni bir eşzamanlı FIFO kuyruk algoritmasının implementasyonunu sunmaktadır. Bu algoritma, Tek Kaynaklı En Kısa Yol (SSSP) problemi için Bellman-Ford algoritmasına uygulanmıştır. Bu çalışmanın temel katkısı, daha önce keşfedilmemiş olan, NVIDIA'nın NVSHMEM teknolojisini kullanan çoklu-GPU concurrent kuyruk sisteminin tasarımı ve implementasyonudur. Bellman-Ford algoritması, önerilen kuyruk sisteminin performansını değerlendirmek amacıyla bir vaka çalışması olarak kullanılmaktadır ve bu çoklu-GPU implementasyonu, bilinen ilk örnek olarak öne çıkmaktadır. Deneysel sonuçlar, çoklu-GPU kuyruk implementasyonunun, dört NVIDIA A100 GPU üzerinde tek-GPU temel performansına kıyasla maksimum 3.92x hız artışı ve ortalama 3.04x hız artışı sağladığını göstermektedir. Bellman-Ford algoritması uygulandığında ise, çoklu-GPU sistemi, tek-GPU implementasyonuna kıyasla maksimum 3.794× hız artışı ve ortalama 3.573× hız artışı elde etmektedir, üretilmiş bir graf ve SuiteSparse Matrix Collection'dan alınan 10 farklı türde graf üzerinde test edilmiştir. Bu bulgular, çoklu-GPU kuyruk sisteminin grafik işleme görevlerinde verimliliğini vurgulamaktadır ve paralel hesaplamadaki pratik zorlukları ele alarak yüksek performanslı hesaplama alanında önemli bir katkı sağlamaktadır.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
MPI	Message Passing Interface
CUDA	Compute Unified Device Architecture
HPC	High Performance Computing
SSSP	Single Source Shortest Path
BF	Bellman-Ford (Algorithm)
FIFO	First-in-First-Out
LIFO	Last-in-First-Out
BQ	Broker Queue
BWD	Broker Work Distributor (Queue)
PGAS	Partitioned Global Adress Space

Chapter 1

INTRODUCTION

Efficient management of data in parallel computing environments is a critical challenge in high-performance computing (HPC). One of the key data structures used in such environments is the queue, which is designed to handle elements in an organized manner. Queues are linear data structures that generally follow one of several principles such as First-In-First-Out (FIFO), Last-In-First-Out (LIFO), or priority-based ordering. The most common form, the FIFO queue, processes elements in the order they arrive, while priority queues allow elements to be processed based on their importance or priority, regardless of their arrival order [Haas et al., 2012] . The choice of queue type depends on the specific needs of the application. Queues are essential for task scheduling, event-driven simulations, and graph algorithms, among others. However, when used in concurrent or parallel environments, ensuring thread-safe operations in the queue becomes a non-trivial challenge. Multiple threads or processes may simultaneously access and modify the queue, requiring careful synchronization to avoid race conditions and ensure data integrity.

The advent of Graphics Processing Units (GPUs) has significantly accelerated parallel computing tasks by providing massive computational power. GPUs, equipped with thousands of smaller, highly parallel cores, are designed to handle large-scale, data-intensive problems more efficiently than traditional CPUs. In particular, NVIDIA's CUDA framework has enabled the development of highly parallel algorithms in areas such as machine learning, simulations, and graph processing. However, implementing efficient concurrent data structures, such as queues, on GPUs is complicated due to the intricacies of memory management and thread synchronization in a parallel execution environment.

While single-GPU implementations have greatly improved the performance of parallel applications, the need for even greater computational power has led to the emergence of multi-GPU systems. In a multi-GPU environment, multiple GPUs work together, each processing a portion of the workload, which allows for the handling of larger datasets and more complex computations. Despite their significant potential, multi-GPU systems introduce challenges related to synchronization, memory coherence, and communication between GPUs. One key solution to address these challenges is NVIDIA's NVSHMEM (NVIDIA Shared Memory) [NVIDIA2, 2024], which facilitates efficient communication and synchronization between GPUs in a multi-GPU setup, allowing for scalable and faster parallel processing. Unlike prior work, which has primarily focused on single-GPU or CPU-based queue implementations, this research explores and demonstrates the efficiency of concurrent queue operations in multi-GPU environments, filling a critical gap in the field.

The Bellman-Ford algorithm, which solves the Single-Source Shortest Path (SSSP) problem, is a fundamental algorithm in graph theory. The SSSP problem involves finding the shortest path from a given source vertex to all other vertices in a weighted graph, where the edge weights may be negative [Bellman, 1958]. Bellman-Ford is particularly useful because it can handle graphs with negative weight edges, but its performance can become a bottleneck in large-scale graphs, particularly when implemented in parallel environments. Despite its importance, there has been limited research into the application of GPU systems for optimizing the Bellman-Ford algorithm, especially with respect to the concurrent queue systems that manage updates to the vertices during the algorithm's execution [Agarwal and Dutta, 2015], [Busato and Bombieri, 2015].

This thesis focuses on the implementation and performance evaluation of multi-GPU concurrent FIFO queue systems, demonstrating significant speedup compared to single-GPU implementations. To showcase the practical application and effectiveness of these queue systems, the Bellman-Ford algorithm was chosen as a case study for solving the Single-Source Shortest Path (SSSP) problem in a multi-GPU environment. Using NVSHMEM for inter-GPU communication, this research aims to fill a critical gap in existing research by optimizing the parallel execution of Bellman-Ford in distributed GPU environments.

Overall, this thesis makes the following contributions:

- **Novel Implementation of Multi-GPU Concurrent FIFO Queues:** A concurrent FIFO queue system optimized for multi-GPU environments is proposed and implemented, addressing the challenges of scalability and synchronization across multiple GPUs. This approach introduces a novel solution, as no prior implementations of multi-GPU concurrent FIFO queues have been explored in the context of parallel graph algorithms. Experimental results demonstrate that the multi-GPU queue implementation achieves a maximum speedup of 3.92× and an average speedup of 3.04× over the single-GPU baseline on four NVIDIA A100 GPUs.

- **Development of Bellman-Ford SSSP with Multi-GPU Queues:** A parallelized Bellman-Ford algorithm utilizing the developed multi-GPU concurrent queue system is designed and implemented. This implementation, being the first to apply multi-GPU concurrent queues to Bellman-Ford, targets the optimization of Single-Source Shortest Path (SSSP) calculations in graph processing tasks. The performance of the multi-GPU Bellman-Ford implementation is thoroughly evaluated and compared across different configurations. The results demonstrate significant improvements in efficiency compared to traditional single-GPU implementations. Extensive testing on a generated benchmark and 10 diverse graphs from the SuiteSparse Matrix Collection shows that the multi-GPU system achieves a maximum speedup of 3.794× and an average speedup of 3.573× compared to the single-GPU implementation.

Chapter 2

BACKGROUND AND MOTIVATION

2.1 Overview

This chapter provides an in-depth examination of GPU architecture and programming models, focusing on CUDA and NVSHMEM. The discussion begins with a comparison of GPUs and CPUs, highlighting the architectural differences and design objectives that make GPUs ideal for parallel workloads. It also introduces key CUDA memory management concepts, including Unified and Global Memory, as well as memory optimization techniques like the L2 Cache Set-Aside. The benefits of Unified Memory in simplifying GPU programming by abstracting memory management are particularly emphasized.

The chapter then transitions to NVSHMEM, a library designed to implement the Partitioned Global Address Space (PGAS) model for multi-GPU systems. NVSHMEM's ability to facilitate low-latency communication make it a powerful tool for high-performance computing on GPU clusters.

In the context of graph algorithms, the chapter explores the importance of the SSSP problem, a cornerstone of graph theory with diverse real-world applications. Various SSSP algorithms, such as Dijkstra's and Bellman-Ford, are discussed, with a particular focus on the latter due to its suitability for parallelization on multi-GPU systems. The Bellman-Ford algorithm's capability to handle graphs with negative edge weights and detect negative cycles makes it a compelling choice for exploring parallel computing techniques [Bellman, 1958]. The chapter concludes by emphasizing the advantages of leveraging multi-GPU systems to accelerate graph algorithms, reduce computation time, and efficiently utilize memory resources.

2.2 Concurrent Queue Algorithms

Blocking and Non-blocking algorithms

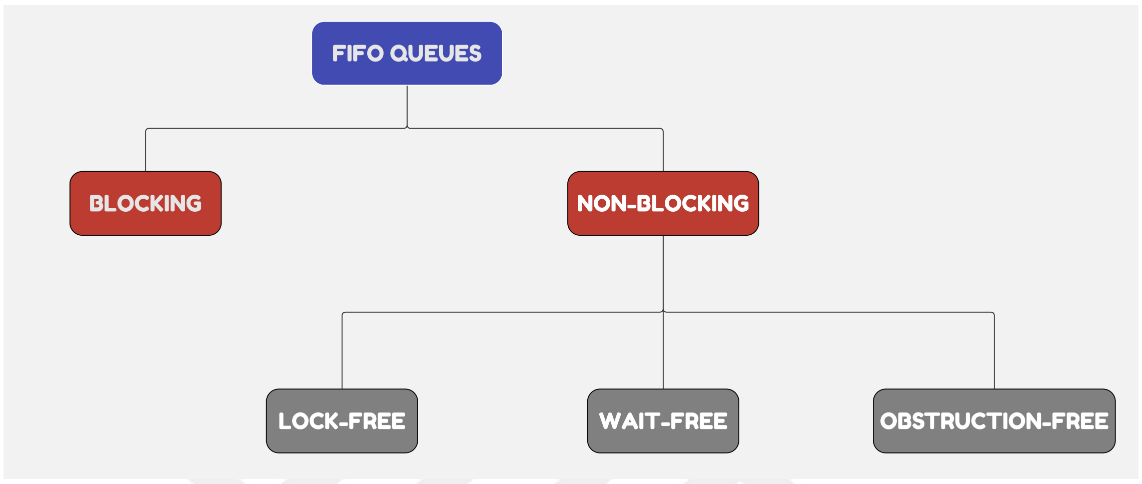


Figure 2.1: Queue Classification

FIFO (First-In, First-Out) queues, characterized by a head for removing items and a tail for adding them, process data items in the exact order they were inserted. For example, if the insertion of item A is completed before the insertion of item B begins, item A must be deleted before item B, ensuring that deletions follow the same sequence as their insertions. This guarantees that the first item to enter is always the first to leave [Gottlieb et al., 1983]. Its inherent ability to maintain tasks in the order of their submission makes it a natural fit for work distribution [Kerbl et al., 2018]. FIFO queues can be further categorized into blocking and non-blocking types [Michael and Scott, 1996]. Blocking algorithms may allow slower processes to delay or even halt the progress of faster ones, potentially causing indefinite delays in completing operations on shared data structures. In contrast, non-blocking algorithms [Stone, 1990b], [Barnes, 1993], [Herlihy, 1993], [Prakash et al., 1994], [Prakash et al., 1991], [Prakash et al., 1994], [Stone, 1992] ensure that at least one active process will complete an operation within a bounded time, even when multiple processes are accessing the structure concurrently.

In asynchronous multiprocessor systems, blocking algorithms face significant performance drops when a process is delayed or stopped. Delays can result from various factors like processor scheduling preemption, cache misses, or page faults. On the other hand, non-blocking algorithms are more resilient, continuing to function effectively even when such disruptions occur. Non-blocking algorithms are designed to ensure that at least one thread in a system will make progress without being indefinitely delayed by others, which contrasts with blocking algorithms where a slow thread can prevent others from completing their operations. However, non-blocking algorithms are not a single monolithic category but rather consist of three distinct types: lock-free, wait-free, and obstruction-free. While these are subcategories of non-blocking algorithms, it is important to understand that not all lock-free algorithms offer the same level of protection against delays or starvation. Even though the queue is lock-free, it can be blocking in some cases [Gottlieb et al., 1983], [Hwang and Faye, 1984], [Mellor-Crummey, 1987], [Sites, 1978], [Stone, 1990a], [Stone, 1990b]. For instance, a slow enqueue operation may prevent dequeue operations attempted by other threads from completing, even though the system as a whole may eventually make progress.

Lock-free algorithms allow individual threads to potentially experience delays or starvation but guarantee that at least one thread will make progress over time. This is in contrast to wait-free algorithms, which ensure that each individual thread will complete its operation within a bounded number of steps, regardless of the actions of other threads [Anthony Williams, 2024]. This means that, in a system using wait-free algorithms, no thread will experience indefinite delays or starvation, making it a more robust solution for high-concurrency environments. All wait-free algorithms are a subset of lock-free algorithms and they provide the strongest guarantee of progress.

”Non-blocking” was used as a synonym for ”lock-free” in the literature until the introduction of obstruction-freedom in 2003 [Herlihy et al., 2003]. Obstruction-free algorithms represent the weakest form of non-blocking guarantees. An algorithm is considered obstruction-free if any thread, when executed in isolation (with no other threads obstructing its progress), will complete its operation within a bounded number of steps. While obstruction-free algorithms provide progress guarantees, they do not offer the same level of protection as lock-free or wait-free algorithms. They allow for the possibility that some threads might experience delays when competing for resources, but they ensure that at least one thread will make progress in such situations. All lock-free algorithms are also obstruction-free, as they guarantee that at least one thread will eventually succeed, but obstruction-free algorithms do not ensure this for every thread involved.

Linearizability and Fair Ordering

Linearizability and fair ordering are critical properties in the design of concurrent data structures, ensuring predictable and correct behavior even in multi-threaded environments.

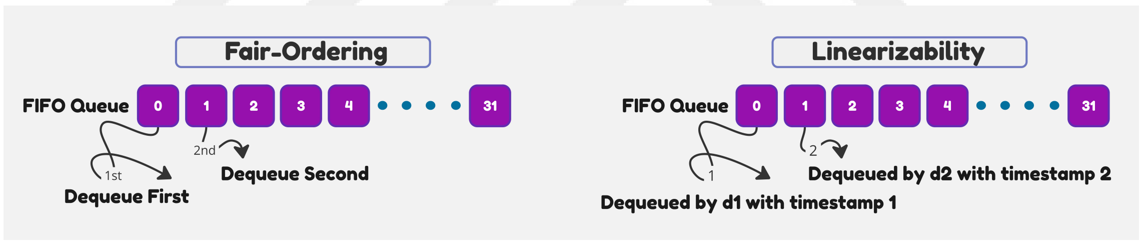


Figure 2.2: Illustration of Linearizability and Fair Ordering in Concurrent Queues.

Linearizability guarantees that operations on a concurrent data structure appear as though they are executed sequentially, respecting the order of their invocation and response events. To verify linearizability, the behavior of the system is modeled as a history, where each method call is represented by a pair of events: an invocation and its corresponding response. Events are considered ordered if the response of one precedes the invocation of another. When two events overlap, meaning their order cannot be determined, the data structure may sequence them arbitrarily, provided it still satisfies the sequential specification. For instance, if one operation dequeues an item at timestamp 1 and another dequeues at timestamp 2, the system ensures this order is observed, even if their actual execution times differ. This property is essential for maintaining logical consistency in concurrent operations [Kerbl et al., 2018].

Fair ordering complements linearizability by ensuring equitable access to shared resources, such as queues. In a FIFO queue, fair ordering guarantees that operations are processed in the order they are queued, preventing starvation or indefinite delays for any single operation. For example, if two operations are enqueued, the first one is always processed before the second, maintaining a predictable and orderly execution sequence. This fairness is particularly important in multi-threaded or distributed systems, where maintaining equal opportunity for access ensures both efficiency and fairness in resource utilization.

GPU Queues

General-purpose computing on GPUs has become increasingly popular for the high-performance, data-intensive applications, leading to a growing demand for efficient data structures to support GPU development. While there has been research on concurrent FIFO queues, studies focusing specifically on GPU-based concurrent FIFO queues remain limited [Zhang et al., 2014], [Cederman et al., 2012], [Misra and Chaudhuri, 2012]. Previous work, such as the research by Misra and Chaudhuri [Misra and Chaudhuri, 2012], demonstrated the use of the CAS operator on GPUs to implement various concurrent data structures. Cederman et al. [Cederman et al., 2012] also explored GPU implementations of the lock-free queue proposed by Michael and Scott. However, both approaches yield performance lower than what is achievable on multi-core platforms.

One key challenge is that these studies treat each thread as an isolated unit for concurrent operations, while modern GPUs schedule tasks in groups of threads (referred to as warps in NVIDIA's terminology). This single-thread-based approach leads to higher contention in CAS operations. As Cederman et al. [Cederman et al., 2012] noted, GPU-based queue operations often lag their multi-core counterparts in terms of speed. To address this limitation, an efficient concurrent lock-free queue, called the Combined and CAS Loop Queue (CCLQ), was developed for GPUs which speed up 40-fold over previous GPU-based implementations.

Selecting a FIFO, concurrent, lock-free, and linearizable queue for implementation on GPUs is essential to achieving high performance and predictable behavior in multi-threaded systems. A FIFO queue ensures that tasks are executed in the precise order of their submission, a critical property for maintaining logical consistency and ensuring fair work distribution across threads. By adopting a concurrent design, the queue can simultaneously handle multiple threads accessing and modifying the data structure, thus maximizing the utilization of GPU resources. The lock-free nature of the queue further enhances performance by guaranteeing that at least one thread can make progress even in the face of contention, preventing the performance bottlenecks that are common in blocking algorithms where thread delays or failures can lead to stalling. Moreover, linearizability guarantees that operations appear to execute atomically, maintaining a globally consistent order that is essential for correctness in complex parallel systems.

These properties make the Broker Queue an ideal choice for implementation on GPUs in this thesis [Kerbl et al., 2018]. Implementing the Broker Queue on GPUs takes full advantage of the massive parallelism that GPUs offer while overcoming significant limitations. These include inefficiencies in managing workloads in massively parallel environments, difficulties in fully utilizing individual threads on single-instruction-multiple-data (SIMD) cores, and the blocking behavior that arises when threads attempt to access the queue simultaneously. This blocking behavior disrupts performance and hinders the queue's application in multi-queue setups, which are crucial for advanced work distribution strategies like work-stealing. The Broker Queue (BQ) was designed to address these challenges by on top of being a highly efficient, fully linearizable FIFO queue optimized for fine-granular parallel work distribution on GPUs. Unlike traditional lock-free queues that rely on optimistic concurrency control and often experience performance degradation due to excessive failed atomic compare-and-swap (CAS) operations, the BQ achieves a balanced approach. It ensures high performance while also maintaining robustness under heavy contention. Additionally, while conventional blocking queues may provide stability, they restrict load balancing by failing to release control back to the calling thread in overflow or underflow scenarios. This limitation makes them unsuitable for multi-queue setups that are crucial for efficient load distribution in parallel systems.

The Broker Queue has been demonstrated to be the highest performing queue implementation in single-GPU environments, as confirmed by extensive performance evaluations [Kerbl et al., 2018]. Given the proven performance of the Broker Queue in single-GPU environments, it is highly likely that its extension to multi-GPU systems will yield even greater performance gains, potentially making this multi-GPU version the fastest queue implementation across all setups.

2.3 GPU Architecture, CUDA and NVSHMEM

Graphics Processing Units (GPUs) are engineered to deliver significantly higher instruction throughput and memory bandwidth compared to Central Processing Units (CPUs) while maintaining comparable cost and power efficiency. This advantage makes GPUs the preferred choice for many applications, allowing them to achieve faster execution times than CPUs. Although other computing devices, such as Field-Programmable Gate Arrays (FPGAs), offer excellent energy efficiency, they lack the programming flexibility that GPUs provide. The fundamental difference between GPUs and CPUs lies in their design objectives. CPUs prioritize maximizing the speed of a single-threaded execution and are optimized for handling a few dozen threads concurrently. In contrast, GPUs are built for massive parallelism, capable of executing thousands of threads simultaneously. This parallelism compensates for the slower performance of individual threads by achieving higher overall throughput. To support this design, GPUs dedicate a larger proportion of their transistors to data processing, minimizing resources allocated to data caching and flow control represented in Figure 2.2. As a result, GPUs are highly specialized for workloads requiring extensive parallel computation, making them ideal for accelerating a wide range of applications [NVIDIA, 2024].

CUDA threads interact with various memory spaces during execution, each designed to optimize performance for specific purposes. Every thread has access to a private local memory (Figure 2.6). Within each thread block, there is shared memory, accessible by all

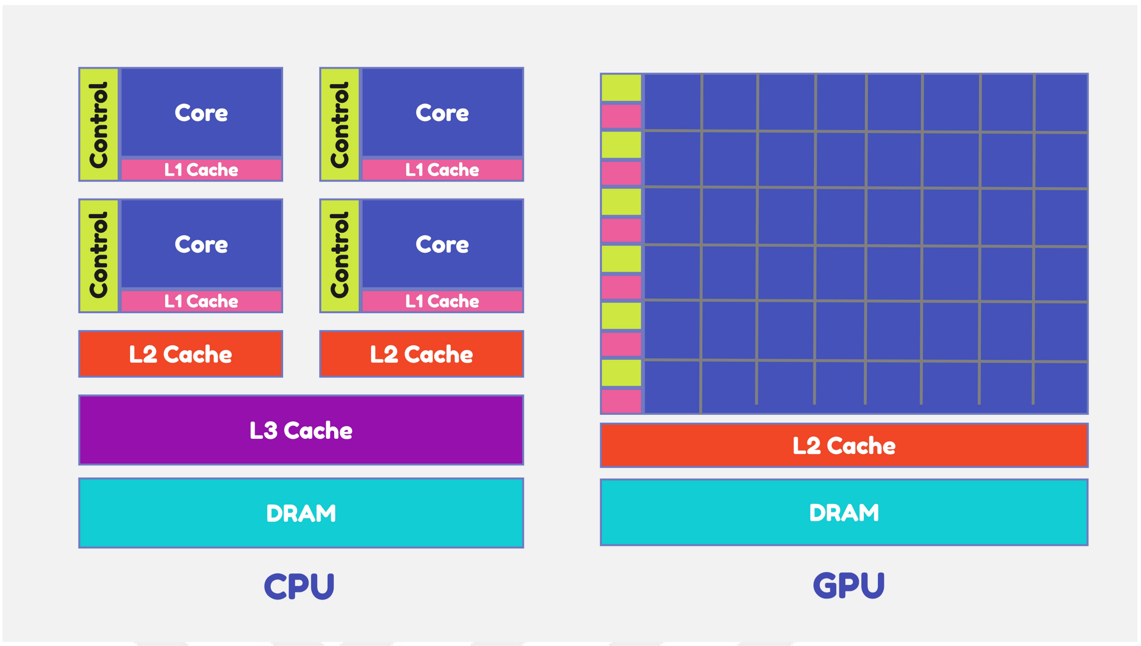


Figure 2.3: CPU vs GPU Architecture

threads in the block, with a lifetime tied to the block's execution. Threads within a thread block cluster can also perform read, write, and atomic operations on the shared memory of other blocks in the cluster. Additionally, all threads can access global memory, which is shared across the entire grid. CUDA provides two read-only memory spaces: constant memory and texture memory which are accessible by all threads. These memory spaces are optimized for specific use cases, with texture memory offering advanced features such as different addressing modes and data filtering for certain data formats. Furthermore, global, constant, and texture memory persist across kernel launches within the same application, ensuring efficient data reuse and management [NVIDIA, 2024]. Morever, the indexing for threads can be visualized as in Figure 2.6 part d.

Unified Memory and Global Memory

Unified Memory provides a shared pool of memory that seamlessly integrates CPU and GPU access, effectively eliminating the divide between the two. With a single pointer, both the CPU and GPU can access this managed memory shown in Figure 2.3, while the

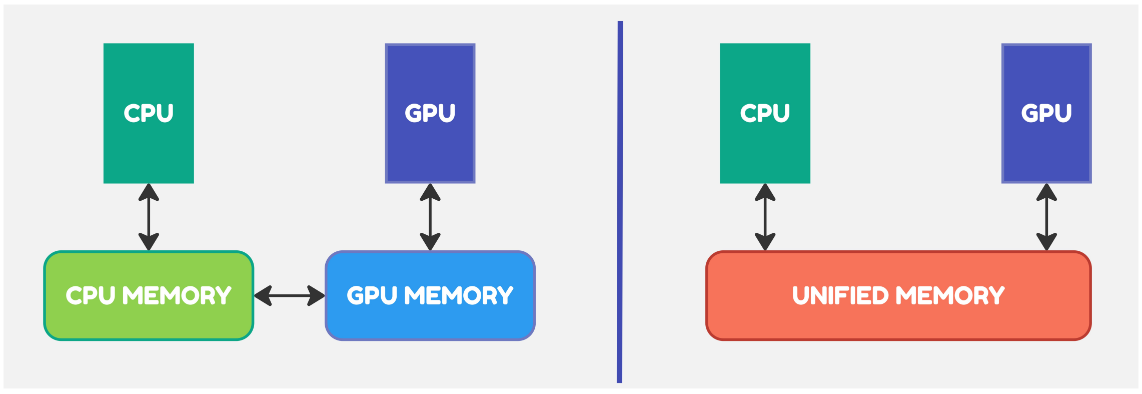
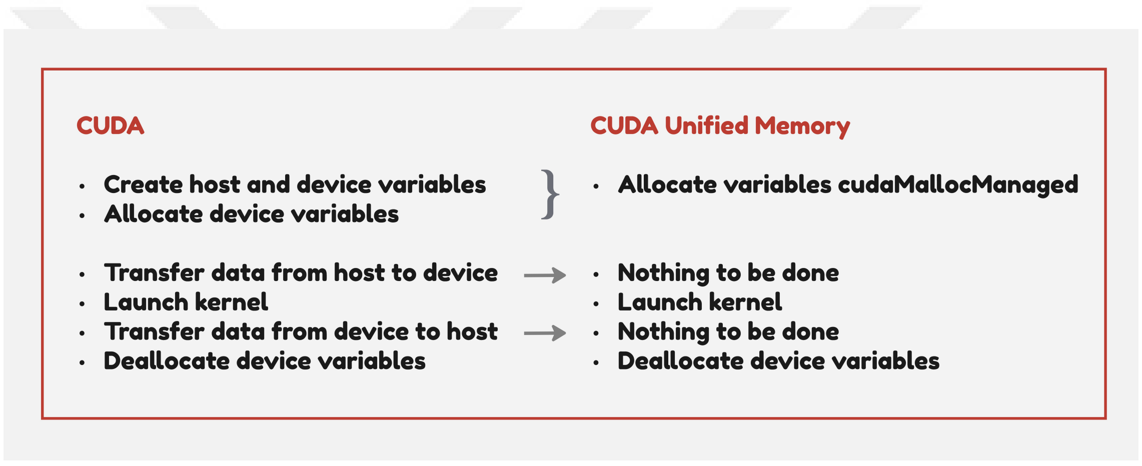


Figure 2.4: Global vs. Unified Memory



CUDA		CUDA Unified Memory
• Create host and device variables • Allocate device variables	}	• Allocate variables cudaMallocManaged
• Transfer data from host to device	→	• Nothing to be done
• Launch kernel		• Launch kernel
• Transfer data from device to host	→	• Nothing to be done
• Deallocate device variables		• Deallocate device variables

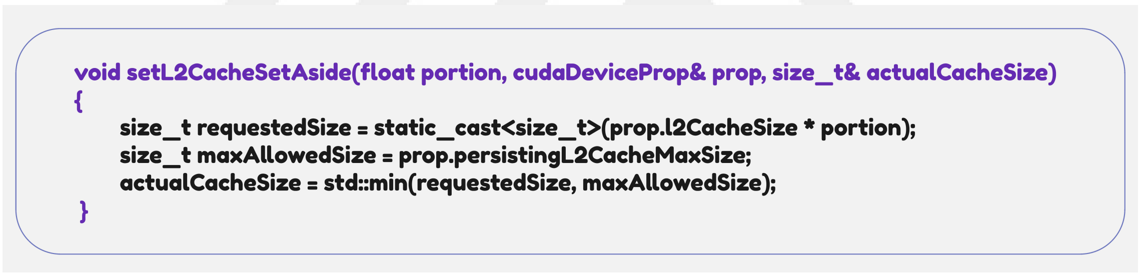
Figure 2.5: Code Related Differences in Global and Unified Memory

system handles the behind-the-scenes data migration between host and device. This approach allows CPU code to treat the memory as standard CPU memory and GPU code to see it as device memory. Figure 2.4 represents the code related differences between Unified and Global Memory. Creation of seperate host and device variables are removed, data transfers are not required in Unified Memory.

By abstracting memory management, Unified Memory simplifies parallel programming on the CUDA platform. It transforms memory allocation and data transfer into optional optimizations rather than mandatory tasks. As a result, developers can focus on designing and implementing CUDA kernels without the overhead of managing device memory. This reduces the complexity of learning CUDA programming and accelerates the process of adapting existing code for GPU execution.

L2 Cache Set-Aside

In CUDA, data accesses to global memory can be categorized based on their usage patterns. When a specific data region is accessed multiple times during kernel execution, these accesses are referred to as persisting. Conversely, streaming accesses occur when data is accessed only once. To optimize performance, a portion of the L2 cache can be reserved specifically for persisting data. This reserved section gives priority to persisting accesses, ensuring they efficiently utilize the cache. In contrast, streaming accesses are only allowed to use this reserved portion if it remains unoccupied by persisting data, thereby minimizing cache contention [NVIDIA, 2024].



```
void setL2CacheSetAside(float portion, cudaDeviceProp& prop, size_t& actualCacheSize)
{
    size_t requestedSize = static_cast<size_t>(prop.l2CacheSize * portion);
    size_t maxAllowedSize = prop.persistingL2CacheMaxSize;
    actualCacheSize = std::min(requestedSize, maxAllowedSize);
}
```

Figure 2.6: L2 Cache Set-Aside

NVSHMEM

Strong scaling, solving a fixed problem faster by increasing processors, is critical for scientific applications but often hindered by CPU-managed inter-node communication in MPI or OpenSHMEM [MPI, 2024]. This reliance results in inefficiencies, such as GPU underutilization and synchronization overheads, particularly for smaller problem sizes. NVSHMEM addresses these challenges by enabling GPU-initiated communication, reducing CPU-GPU synchronization, and unlocking GPU parallelism. Built on the OpenSHMEM specification, NVSHMEM simplifies development by abstracting intricate low-level configurations typically necessary for efficient communication. It allows direct memory access and manipulation across processes, ensuring low-latency operations required for multi-GPU computation. By providing a unified global address space that spans the memory of multiple GPUs, NVSHMEM supports flexible access patterns, including GPU-initiated operations, CPU-initiated actions, and operations integrated with CUDA streams.

NVSHMEM introduces an advanced programming abstraction tailored for NVIDIA GPU clusters, implementing the Partitioned Global Address Space (PGAS) model as shown in Figure 2.6. This experimental library, developed by NVIDIA, enhances performance and reduces programming complexity in large-scale GPU environments [Hsu et al., 2020]. By eliminating frequent CPU synchronization, NVSHMEM allows developers to create long-running kernels that overlap communication and computation by leveraging GPU thread warp scheduling. This reduces overheads associated with kernel launches, CPU-GPU synchronization, and API calls. Additionally, NVSHMEM provides the flexibility of CPU-side communication for scenarios where GPU-initiated communication may not suffice, ensuring broad applicability [NVIDIA2, 2024].

It should be taken into account that even though the variables are in the shared PGAS, when PEs work on them, only the related PE's copy of the variable value will be changed. For example, in Figure 2.6 part a shows that value is set to 0 in all PEs, when PE 1 sets the value to 1, only PE 1's memory is updated, the rest of value variables in other PEs stayed as 0. However, the partitioned address space in NVSHMEM offers a significant advantage: all other PEs can access and update the modified variable of PE0 as needed. This is made possible through NVSHMEM's one-sided communication model, which leverages put/get APIs to enable efficient remote data access and seamless data movement [NVIDIA2, 2024].

Traditional inter-GPU communication methods, such as the Message Passing Interface (MPI), struggle with fine-grained, concurrent GPU interactions due to synchronization requirements between send and receive operations [MPI, 2024]. These limitations lead to inefficiencies, including locking overheads, serialization delays, and protocol mismatches. NVSHMEM's one-sided communication primitives address these challenges by allowing initiating threads to define all transfer parameters, eliminating sender-receiver synchronization. These primitives, efficiently mapped to RDMA or NVLink load/store operations, simplify application design and improve performance by enabling computation and communication overlap—an essential feature for scaling GPU workloads.

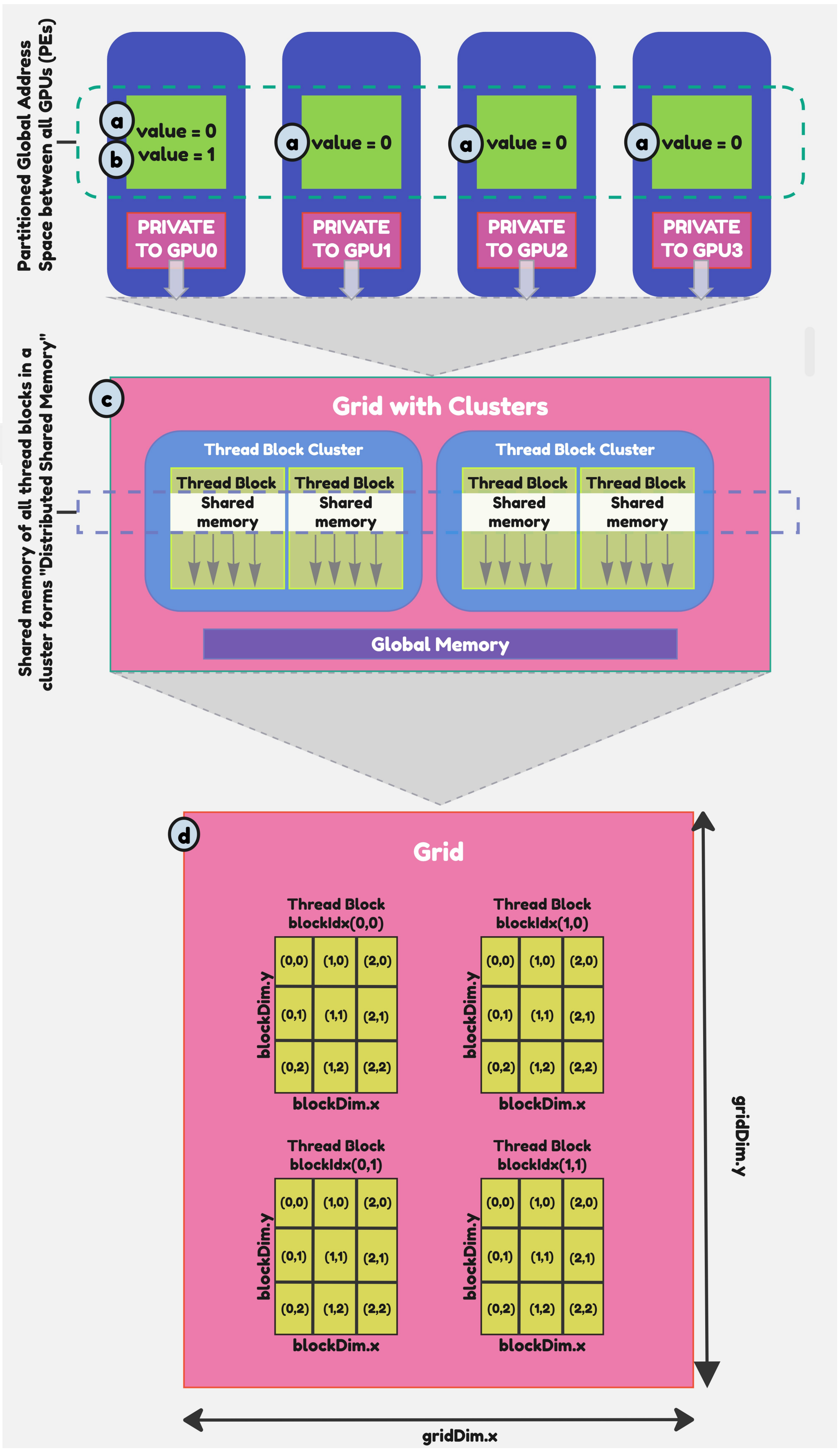


Figure 2.7: CUDA and NVSHMEM

2.4 SSSP Algoritms - Bellman Ford

With advancements in computer and information technology, there has been a growing interest in research focused on graph algorithms. The shortest-path problem, a fundamental topic in computer science and graph theory, seeks an optimal path with the minimum length between a source and a destination. This problem has diverse range of applications including network routing, planning of possible routes, urban transportation systems, pathfinding in social networks, and so on [Madkour et al., 2017].

Shortest-path algorithms are typically categorized into two main types: single-source shortest-path (SSSP) and all-pairs shortest-path (APSP). The SSSP problem focuses on finding the shortest paths from a single source vertex to all other vertices in a graph, whereas APSP aims to compute the shortest paths between every pair of vertices [Madkour et al., 2017].

Formally, the SSSP problem can be defined as follows: given a graph $G = (V, E)$ and a source vertex $s \in V$, compute the distances $\delta(s, v)$ for all $v \in V$. In unweighted graphs, the problem can be efficiently solved using a breadth-first search (BFS) [Cormen et al., 2022]. Starting from the source vertex, BFS explores all adjacent vertices, expanding the search until the shortest path with the minimum number of edges to each destination vertex is identified.

For graphs with weighted edges, Dijkstra's algorithm is a widely used solution for the SSSP problem when edge weights are non-negative. This algorithm classifies vertices into two groups: solved and unsolved. Initially, the source vertex is considered solved, and the algorithm iteratively selects the shortest edge connecting the solved vertices to the unsolved ones. After processing the selected edge, the connected vertex is added to the solved set. This process repeats until all vertices are solved. Dijkstra's algorithm avoids examining all edges, making it efficient for graphs where certain edges have high computational costs. However, its limitations include being applicable only to static graphs with non-negative weights. The algorithm has a time complexity of $O(n^2)$ and cannot detect or handle negative weight cycles [Madkour et al., 2017]. Additionally, while Dijkstra's algorithm requires numerous iterations and presents significant challenges for parallel implementation due to its sequential nature, the Bellman-Ford algorithm offers superior parallelization opportunities with fewer iterations, as each iteration can be effectively executed in parallel. [Chakaravarthy et al., 2016]

The Bellman-Ford algorithm, introduced by Bellman, Ford, and Moore [Bellman, 1958], [Ford, 1956], [Moore, 1959], addresses these limitations by offering a more versatile approach that can handle graphs with negative edge weights. Unlike Dijkstra's algorithm, Bellman-Ford processes all neighboring edges of a vertex rather than just the shortest one. It operates by iteratively relaxing the edges of the graph, allowing it to find the shortest paths even in the presence of negative-weight edges. The algorithm begins by initializing the distance to the source vertex as zero and all other vertices as infinity. Then it repeatedly examines each edge, updating the distance to the destination vertex if a shorter path is found through the source vertex. This process is repeated for a number of iterations equal to the number of vertices minus one, ensuring that all possible paths are considered. After completing the iterations, the algorithm checks for negative cycles, which can indicate that further relaxation is possible, thus confirming the presence of a negative weight cycle in the graph. The Bellman-Ford algorithm has a time complexity of $O(nm)$, where n is the number of vertices and m is the number of edges. Its strengths lie in its ability to accommodate negative weights and detect negative cycles, but it is slower compared to Dijkstra's algorithm and continues to iterate even when further updates to the graph weights are unnecessary.

While Dijkstra's algorithm strictly requires a priority queue to maintain the ordering of vertices based on their tentative distances, the Bellman-Ford algorithm offers more flexibility in its data structure choice [Chen et al., 2007]. Although it can be implemented with a simple array structure, using a FIFO queue can optimize its performance by better managing the order of vertex processing. This compatibility with FIFO queues, combined with its inherent parallelizability and ability to handle negative weight cycles, makes the Bellman-Ford algorithm an ideal candidate for implementing and testing the concurrent

FIFO queue system proposed in this thesis. Utilizing parallel computing to address the SSSP problem through Bellman-Ford can yield two key benefits: it can significantly reduce computation time compared to sequential methods and leverage the combined memory resources to mitigate the limitations of slow external memory operations.

Currently, traditional serial graph algorithms face time constraints due to their inefficiency, often requiring substantial processing time. As a result, parallel computation emerges as a viable solution to enhance performance. By imposing certain constraints on the data and capitalizing on the capabilities of modern hardware, parallel approaches can effectively optimize the execution of graph algorithms [Agarwal and Dutta, 2015].

Chapter 3

METHODOLOGY AND IMPLEMENTATION

3.1 Single-GPU Queue Implementation

The broker queue is a concurrent data structure designed for efficient and reliable management of enqueue and dequeue operations, particularly in parallel environments [Kerbl et al., 2018]. It comprises four main components:

1. Ring Buffer:

A circular data structure where elements are stored in a fixed-size buffer. Once the buffer is full, it wraps around to overwrite older elements.

2. Head and Tail Pointers:

These pointers are used for ticketing and tracking the positions of enqueue and dequeue operations. Elements are added to the tail and removed from the head of the ring buffer.

3. Ticket Buffer:

Ensures proper sequencing by locking individual queue slots, preventing conflicts in concurrent operations.

4. Counter (Count):

This counter maintains the balance between the number of enqueue and dequeue operations, ensuring consistency even when multiple threads operate concurrently.

This queue algorithm does concurrent enqueue and dequeue operations on a circular ring buffer. However, the importance of this algorithm is 1) Ensuring possible future enqueue and dequeue operations by holding a count variable and guaranteeing linearizability 2) Ticketing mechanism for access to prevent possible race conditions and guaranteeing fair-ordering.

1) Ensuring:

In traditional queue implementations, when a thread attempts to enqueue an item, it will typically skip the operation if the queue is full and there is no available space. This behavior is a common limitation of standard queue structures. However, the strength of this implementation lies in its novel approach to handling this issue. Specifically, the queue introduces a ”broker” mechanism within its interface. This mechanism not only tracks the items stored in the ring buffer but also guarantees the successful execution of enqueue or dequeue operations before they are performed, hence the name ”broker queue.”

To perform an enqueue or dequeue operation, a thread must first obtain a guarantee, or assurance, that it will be able to complete the operation. This is done through an additional counter variable, referred to as ”Count” which plays a crucial role in ensuring the integrity of the queue. The purpose of this counter is to prevent any thread from modifying the head or tail pointers unless it is assured that its operation will be successfully completed.

For enqueue operations, the method ‘ensureEnqueue‘ is used to provide this assurance. It returns ‘true‘ if there is sufficient space available in the ring buffer to store the item, or if other threads have already committed to dequeuing items, thereby freeing up space. Similarly, for dequeue operations, the ‘ensureDequeue‘ method returns ‘true‘ if there is an item available in the ring buffer to be dequeued, or if at least one thread has committed to enqueuing an item that will soon become available for dequeuing.

By utilizing this broker mechanism, the system ensures that only threads that have received confirmation that their operation will be successfully executed are allowed to modify the critical queue pointers (head and tail). The design significantly improves the reliability of concurrent enqueue and dequeue operations, particularly in environments where multiple threads are accessing the queue simultaneously. This ensuring mechanism is what makes the Broker Queue linearizable, as it guarantees a consistent and correct order of operations, preserving the integrity of the queue even in highly concurrent scenarios.

2) Ticket Buffer and Ticketing Mechanism:

To prevent race conditions and ensure orderly access to the queue, algorithm incorporates a ticketing mechanism. Each slot in the ring buffer is associated with a ticket, and enqueue and dequeue operations are serialized based on the ticket number. The ticket buffer stores the ticket values, which are used to control the order in which threads can access the queue. Even-numbered tickets are assigned to enqueue operations, while odd-numbered tickets are assigned to dequeue operations. When a thread attempts to perform an enqueue or dequeue operation, it must first obtain a ticket corresponding to the slot it wishes to access. The thread then waits for its ticket number to match the one stored in the ticket buffer before proceeding with its operation. This ensures that enqueue and dequeue operations are executed in a controlled, sequential order, preventing race conditions and ensuring fair-ordering property in the queue's state [Kerbl et al., 2018].

Algorithm 1 Single-GPU Broker Queue Algorithm

```
Function INITIALIZE ():
    $head \leftarrow 0$;
    $tail \leftarrow 0$;
    $count \leftarrow 0$;
    $Tickets[N] \leftarrow \{0, 0, \ldots, 0\}$;
    $RingBuffer[N] \leftarrow \{0, 0, \ldots, 0\}$;

Function WAIT_FOR_TICKET (Pos, Expected):
    while $Ticket[Pos] \neq Expected$ do
        sleep()
    end

Function ENSURE_ENQUEUE ():
    $Num \leftarrow$ atomicLoad(count);
    $ensurance \leftarrow$ false;
    while $Num < N$ and !ensurance do
        if atomicAdd(count,1) $< N$ then
            $ensurance \leftarrow$ true
        end
        else
            $Num \leftarrow$ atomicSub(count,1) $-$ 1;
        end
    end
    return ensurance;

Function ENSURE_DEQUEUE ():
    $Num \leftarrow$ atomicLoad(count);
    $ensurance \leftarrow$ false;
    while $Num > 0$ and !ensurance do
        if atomicAdd(count, -1) $<= 0$ then
            atomicAdd(count,1);
            $Num \leftarrow$ atomicLoad(count);
        end
    end
    return ensurance;

Function PUT_DATA (data):
    $Pos \leftarrow$ atomicAdd(tail,1);
    $P \leftarrow$ Pos % N;
    WAIT_FOR_TICKET(P, 2*(Pos/N));
    $RingBuffer[P] \leftarrow$ data;
    $Tickets[P] \leftarrow$ 2*(Pos/N) + 1;

Function READ_DATA (data):
    $Pos \leftarrow$ atomicAdd(head,1);
    $P \leftarrow$ Pos % N;
    WAIT_FOR_TICKET(P, 2 * (Pos / N) + 1);
    poppedData $\leftarrow$ RingBuffer[P];
    $Tickets[P] \leftarrow$ 2*(Pos + N/N);

Function ENQUEUE (data):
    while not ENSURE_ENQUEUE() do
        if $N \leq (tail - head) < N + MaxThreads/2$ then
            return Full;
        end
    end
    PUT_DATA(data);
    return Successful;

Function DEQUEUE (data):
    while not ENSURE_DEQUEUE() do
        if $N + MaxThreads/2 \leq (tail - head - 1)$ then
            return Empty;
        end
    end
    return READ_DATA(data);
```

There is another version of Broker Queue implemented and tested in this thesis is called ”Broker Work Distributor Queue (BWD)”. The shift from a broker queue to a broker work distributor is quite simple. Instead of continuously checking the ensureEnqueue and ensureDequeue functions in a loop to verify the Full or Empty status, these functions are called just once during the enqueue and dequeue processes. The result of this single invocation is taken at face value, indicating Full or Empty if the broker cannot find an available slot or match right away.

One limitation of the Broker Work Distributor (BWD) is its non-linearizability. Since the Count variable serves merely as a temporary assurance, it does not accurately reflect the actual state of the queue when items are added or removed. While this may be a concern for a queue that needs to operate strictly as a concurrent FIFO structure, it is generally acceptable in the context of distributing work. If an ensureEnqueue call returns false, it implies that, according to all threads that have interacted with the queue up to that point, all items will be drained; that is, unless another thread enqueues, the queue will eventually become Empty. This behavior is arguably sufficient for work distribution and provides a practical indication for efficiently transitioning to another queue that may already have tasks available.

3.2 Multi-GPU Queue Implementation

In a traditional single GPU setup, all enqueue and dequeue operations are performed on a centralized queue. This can lead to significant performance limitations, particularly when multiple processes attempt to access the queue simultaneously, resulting in contention and delays. The primary objective of this thesis is to evaluate the performance improvement achieved by transitioning from a single GPU centralized queue to a multi-GPU distributed queue where each GPU to manage its own portion of the queue. This approach leverages the capabilities of multiple GPUs to enhance throughput and reduce latency in processing queue operations. By distributing the workload across several GPUs as in Figure 3.1, the system can handle a higher volume of operations concurrently, thereby minimizing the bottlenecks typically associated with a single-threaded queue.

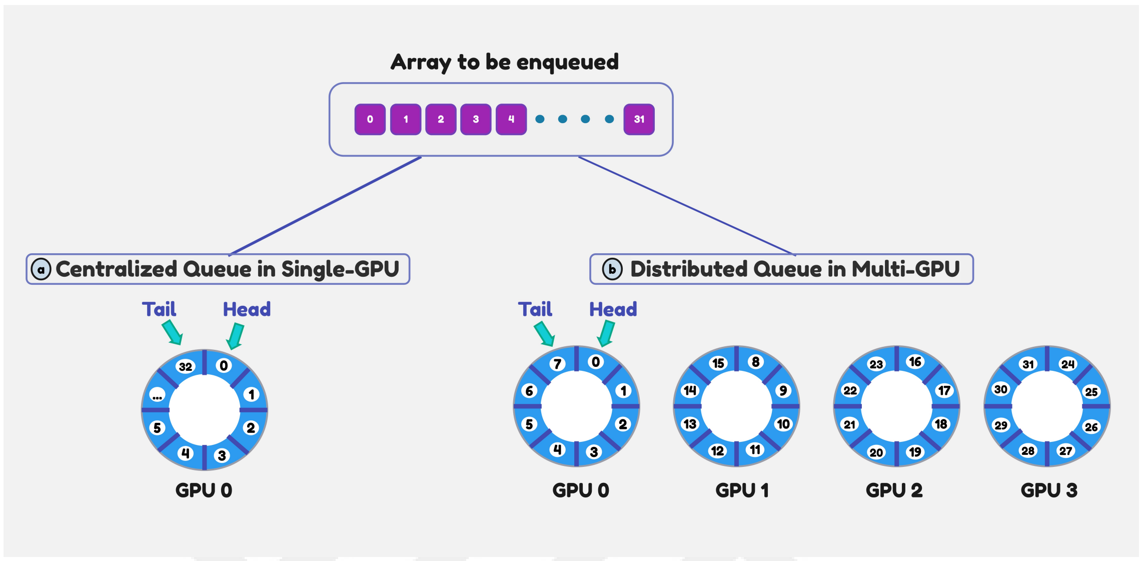

Array to be enqueued
0 1 2 3 4 31
ⓐ Centralized Queue in Single-GPU
ⓑ Distributed Queue in Multi-GPU
Tail
Head
GPU 0
GPU 0
GPU 1
GPU 2
GPU 3

Figure 3.1: Illustration of Single-GPU and Multi-GPU scenarios

The implementation utilizes a Partitioned Global Address Space (PGAS) model, where each GPU creates its own ring buffer, ticket buffer, and head and tail variables. These components are shared among all GPUs, enabling them to coordinate their operations effectively. The queue is partitioned based on the number of GPUs as it can be seen in Figure 3.1 scenario b, with each GPU storing a fraction of the total queue size. This coalesced storage approach allows for efficient memory usage and ensures that each GPU can enqueue and dequeue items from its distributed queue without interference from others.

By enabling each GPU to handle its own queue operations, the workload is effectively divided and parallelized. This parallelization not only speeds up the overall processing time but also enhances the system's ability to scale with the addition of more GPUs. As a result, the multi-GPU distributed queue architecture represents a significant advancement over the traditional single GPU model, providing a robust solution for high-performance computing applications that require efficient data handling and processing.

Moreover, it is important to recognize that when kernels are initially executed in a CUDA environment, there is typically an overhead associated with kernel launch and execution [NVIDIA, 2024]. This overhead can arise from various factors, including the time required for the GPU to transition from a low-power state, the compilation of the kernel code, and the initialization of memory resources. To mitigate the impact of this overhead on performance measurements, a warm-up phase is implemented prior to the commencement of timing for the tests. Specifically, all kernels intended for use are executed ten times to ensure that they are fully optimized and that any initial overhead is accounted for. Additionally, the data transfer overhead associated with the array from which elements will be enqueued is completed before the timing begins. Consequently, only the enqueue and dequeue operations are subjected to performance testing, providing a more accurate assessment of the algorithm's efficiency.

3.3 Single and Multi-GPU Bellman-Ford

The Bellman-Ford algorithm is a well-known method for finding the shortest paths from a single source vertex to all other vertices in a weighted graph. Single GPU implementation is leveraging CUDA to parallelize the workload and enhance performance. The primary goal is to efficiently handle the graph's edges and compute the shortest distances while managing potential negative cycles.

The implementation begins by reading the graph data from a given input file, which includes the number of vertices and edges, as well as the individual edges themselves. This data is stored in a vector of Edge structures, which encapsulates the source vertex, destination vertex, and the weight of the edge. The graph is then transferred to the GPU memory for processing. The distances from the source vertex to all other vertices are initialized in a device array, where the distance to the source is set to zero, and all other distances are initialized to infinity.

To facilitate the parallel processing of edges, the broker queue (BQ) structure presented in this thesis is employed. This queue is specifically designed to hold the edges that require relaxation during each iteration of the Bellman-Ford algorithm. By allowing multiple threads to enqueue edges concurrently, the implementation significantly enhances the efficiency of the algorithm. This parallelization is crucial, as it reduces the time spent on processing each edge, enabling faster computations.

The core logic of the Bellman-Ford algorithm involves dequeuing edges from the queue and attempting to relax them. When a thread retrieves an edge, it first checks if the distance to the source vertex of that edge is not infinity. If this condition is met, the algorithm calculates a new potential distance to the destination vertex. Should this new distance be shorter than the currently known distance, the algorithm updates the distance and increments a change counter. This counter plays a vital role in determining whether any updates were made during the iteration, which is essential for detecting negative cycles in the graph.

Throughout the execution, the algorithm iterates over the vertices, resetting the queue and preparing the edges for relaxation in each cycle. After a specified number of iterations, it checks for any changes made in the last iteration to identify the presence of negative cycles. The results, including execution time, the number of iterations, and the shortest distances from the source vertex, are then printed to the console for analysis.

In summary, this single-GPU implementation of the Bellman-Ford algorithm effectively utilizes CUDA to parallelize edge processing, leading to significant performance improvements compared to traditional CPU-based approaches. By leveraging a queue structure and managing distances in GPU memory, the implementation efficiently computes the shortest paths while adeptly handling the complexities associated with negative cycles.

It is worth noting that this implementation is not the most optimized version of the single-GPU Bellman-Ford algorithm. Prior studies in the literature, developed by larger teams over extended periods, have resulted in highly optimized implementations that achieve superior performance [Zhang et al., 2020]. The purpose of this work, however, was not to create the fastest single-GPU Bellman-Ford algorithm. Instead, the focus was on evaluating the performance of a queue-based implementation on a single GPU and analyzing the scalability and speedup when transitioning to a multi-GPU environment. This approach provides valuable insights into the performance trends of queue-based shortest-path algorithms in multi-GPU setups. The pseudocode of the single-GPU Bellman-Ford algorithm can be found below.

Algorithm 2 Bellman-Ford Algorithm

```
Input: Graph $G$ with vertices $V$ and edges $E$, source vertex $s$
Output: Shortest distances $d$ from source vertex, detection of negative cycles
Function INITIALIZEDISTANCES (Graph G, Source s):
    for each vertex $v$ in $G.V$ do
        if $v == s$ then
            $d[v] \leftarrow 0$
        end
        else
            $d[v] \leftarrow \infty$
        end
    end
Function RELAXEDGES (Graph G):
    for each edge $(u, v)$ in $G.E$ do
        if $d[u] + weight < d[v]$ then
            $d[v] \leftarrow d[u] + weight$
        end
    end
Function BELLMANFORD (Graph G, Source s):
    INITIALIZEDISTANCES (G, s)
    $hasNegativeCycle \leftarrow$ false
    for int $i = 0$ to $V - 1$ do
        RELAXEDGES (G)
        if $i == V - 1 \wedge changes > 0$ then
            $hasNegativeCycle \leftarrow$ true
        end
    end
    return $d, hasNegativeCycle$
```

Multi-GPU Bellman-Ford Implementation:

The multi-GPU implementation of the Bellman-Ford algorithm introduces several significant enhancements compared to the single-GPU version, primarily aimed at leveraging the capabilities of multiple GPUs to improve performance and efficiency in processing large graphs. One of the most notable changes is the integration of NVSHMEM for inter-GPU communication and MPI for process management. This combination allows multiple GPUs to collaborate on the same problem, facilitating data sharing and synchronization of operations. By incorporating nvshmem and mpi.h, the implementation enables different processing elements (PEs) to coordinate their efforts effectively, which is crucial for executing the Bellman-Ford algorithm in a parallelized manner. In this multi-GPU version, the graph data is initially read by the rank 0 process, which then broadcasts the information to all other processes using MPI. This ensures that each GPU has access to the same graph data, a critical requirement for parallel processing. The edges are allocated in NVSHMEM memory, allowing all GPUs to access the same data without the need for explicit data transfers between them. This approach not only simplifies data management but also enhances the overall efficiency of the algorithm by reducing the overhead associated with data movement.

The core logic of the algorithm is modified to enable parallel edge processing. The bellmanFordRelax kernel is designed to divide the workload among the available GPUs, with each GPU processing a specific portion of the edges based on its rank. This division of labor allows for concurrent processing, significantly speeding up the relaxation step of the algorithm. Each GPU calculates its start and end indices for edge processing, ensuring comprehensive coverage of all edges without overlap. To maintain data integrity during distance updates, the multi-GPU implementation employs NVSHMEM atomic operations. These operations ensure that updates to the distance array are performed safely across multiple GPUs, preventing race conditions when multiple threads attempt to update the same distance values.

Additionally, the implementation includes a mechanism for detecting changes across GPUs after each iteration. If any changes are detected during the last iteration, the algorithm can conclude that a negative cycle exists. This collective approach to change detection is vital for ensuring the accuracy of the Bellman-Ford algorithm when executed in a multi-GPU setup.

Finally, the multi-GPU version incorporates performance measurement tools to evaluate the algorithm's efficiency across various configurations of thread and block counts. The results, including execution time, number of iterations, and detection of negative cycles, are collected and written to a CSV file for further analysis. This comprehensive evaluation allows for a detailed comparison of the multi-GPU implementation's performance against the single-GPU version, highlighting the benefits of parallel processing in handling complex graph algorithms.

In the implementation of the Bellman-Ford algorithm across multiple GPUs, a significant optimization is achieved through the strategic distribution of the distance array among the available GPUs. This distribution is crucial for minimizing overhead and enhancing parallelism during the relaxation phase of the algorithm.

During the Bellman-Ford relaxation process, each GPU is responsible for accessing and updating the distance values associated with the vertices it processes. By distributing the distance array, each GPU can directly access the distance values relevant to the vertices it is handling. This approach is particularly important because it reduces the need for inter-GPU communication, which can introduce substantial latency and overhead.

If the distance array were centralized on a single GPU, any access to the distance values would require communication between GPUs. This inter-GPU communication can significantly degrade performance, especially as the number of GPUs increases, due to the time taken for data transfers across the interconnect. In contrast, by partitioning the distance array, each GPU can operate on its local copy of the relevant distance values, allowing for faster access and updates.

This design choice not only improves the performance of the Bellman-Ford algorithm in a multi-GPU environment but also adheres to the principles of parallel computing. By reducing contention for shared resources and maximizing local data access, the implementation achieves higher efficiency. Consequently, the algorithm becomes more scalable, effectively leveraging the computational power of multiple GPUs while mitigating the performance penalties typically associated with distributed memory architectures.

Moreover, another significant optimization arises from the choice of atomic operations used for managing the distributed queues. Instead of relying on the more complex ”nvshmem_TYPENAME_atomic_compare_swap” operations, which are designed for scenarios where multiple threads or processes may contend for the same memory location, the implementation leverages simpler atomic operations such as atomicAdd.

Given that the queues are distributed across the GPUs, each GPU operates on its own local version of the queue. It is important to note that this architecture still enables each GPU to access queue contents on other GPUs whenever necessary. As a result, each GPU primarily writes to its own queue without the necessity of synchronizing with other GPUs for every operation.

This optimization is feasible because, when a GPU is working within its own Partitioned Global Address Space (PGAS), the NVSHMEM library's definitions reveal that it primarily utilizes simple atomic operations, such as atomicAdd, for managing updates. This means that the overhead typically associated with more complex atomic operations, like NVSHMEM's compare-and-swap mechanism, is avoided. By employing atomicAdd and similar atomic operations, the implementation can efficiently update the queue without incurring the performance penalties that would arise from using more intricate synchronization methods. This choice not only enhances the performance of the Bellman-Ford algorithm in a multi-GPU environment but also exemplifies a key strategy in optimizing parallel algorithms by leveraging the capabilities of the NVSHMEM library effectively. It is important to note that when updates to data residing on other GPUs are necessary, one must utilize the one-sided communication mechanisms provided by the NVSHMEM library, such as put and get. These mechanisms are essential during the relaxation steps to access and compare the distance values of two different vertices that are distributed across two separate GPUs, corresponding to their respective vertex numbers.

Chapter 4

EXPERIMENTAL RESULTS AND EVALUATION

To evaluate the performance of the novel multi-GPU queue implementation, a series of tests were conducted in a high-performance computing environment with 4 NVLink-connected NVIDIA A100-SXM4-40GB GPUs, running CUDA version 12.3. The experimental evaluation has two comprehensive phases of performance analysis. In the first phase, the concurrent queue implementation undergoes extensive testing across multiple configurations. The single-GPU evaluation examines performance characteristics under varying parameters: thread counts (32-512), block counts (64, 108, 216), L2 cache set-aside ratios (0-75 percent), and both unified and global memory implementations. Subsequently, the multi-GPU evaluation focuses on global memory configurations without L2-cache set-aside, analyzing performance across different thread and block count combinations. The comparative analysis between single and multi-GPU implementations yields significant speedup metrics, quantifying the benefits of the distributed approach. The second phase validates the implementation's practical efficacy through the Bellman-Ford Single-Source Shortest Path (SSSP) algorithm as a case study. This phase involves systematic testing of both single and multi-GPU implementations across various thread and block configurations. The performance comparison between single and multi-GPU Bellman-Ford implementations obtained by speedup provides crucial insights into the efficiency of the proposed queue system in real-world graph processing applications. The tests are performed on both a generated benchmark and ten distinct category graphs sourced from the SuiteSparse Matrix Collection. [SuiteSparse, 2024]. Additionally, the speedup when 2 GPUs utilized instead of 4 GPUs are discussed under Section 4.2.

1. Testing Concurrent FIFO Queue Implementation:

a. Single GPU Performance Tests:

Two different queue versions, Broker and BWD, were tested under varying configurations:

- **Thread Counts:** 32, 64, 128, 256, 512
- **Block Counts:** 64, 108, 216
- **L2 Cache Set-Aside Ratios:** 0%, 25%, 50%, 75%
- Tests were conducted on both unified memory and global memory to assess the impact of different memory configurations.

For Single-GPU tests, 10 million concurrent operations are performed on the queue. In the Global Enqueue-Dequeue (EnqDeq) scenario, the focus is on sequentially performing enqueue operations followed by dequeue operations. This approach allows for a clear assessment of the system's performance when handling these operations in isolation. Conversely, the Global Mixed scenario is designed to evaluate the concurrent performance of both enqueue and dequeue operations. In this setup, odd-numbered blocks are designated for dequeue operations, while even-numbered blocks are responsible for enqueueing. This arrangement simulates a more realistic workload, where multiple threads interact with the queue do perform different operations simultaneously, providing valuable insights into the system's efficiency and responsiveness under concurrent conditions.

b. Multi-GPU Performance Tests:

The multi-GPU tests focused on global memory configurations, with no cache set aside:

- **Thread Counts:** 32, 64, 128, 256, 512
- **Block Counts:** 64, 108, 216

2. Testing Bellman-Ford Algorithm:

Firstly, the tests were performed on a generated graph containing negative weights, with 11,000 vertices and 12,005 edges, to simulate real-world graph-based workloads. Then, 10 different kinds of graphs were taken from SuiteSparse Matrix Collection [SuiteSparse, 2024] to test the Multi-GPU vs Single-GPU speedup. Performance tests were also conducted using the Bellman-Ford algorithm both **Single-GPU and Multi-GPU**:

- **Thread Counts:** 32, 64, 128, 256, 512
- **Block Counts:** 64, 108, 216
- Only global memory was used, with no cache set aside, to measure the performance impact of the multi-GPU queue implementation on graph traversal tasks.

These comprehensive tests were designed to capture a wide range of performance metrics and ensure that the multi-GPU queue implementation can scale effectively across different configurations and workloads. The results and analysis are explained in the below.

4.1 Single-GPU Queue Implementation Results

4.1.1 Unified vs Global Memory with Thread and Block Count Variations

Single GPU Tests

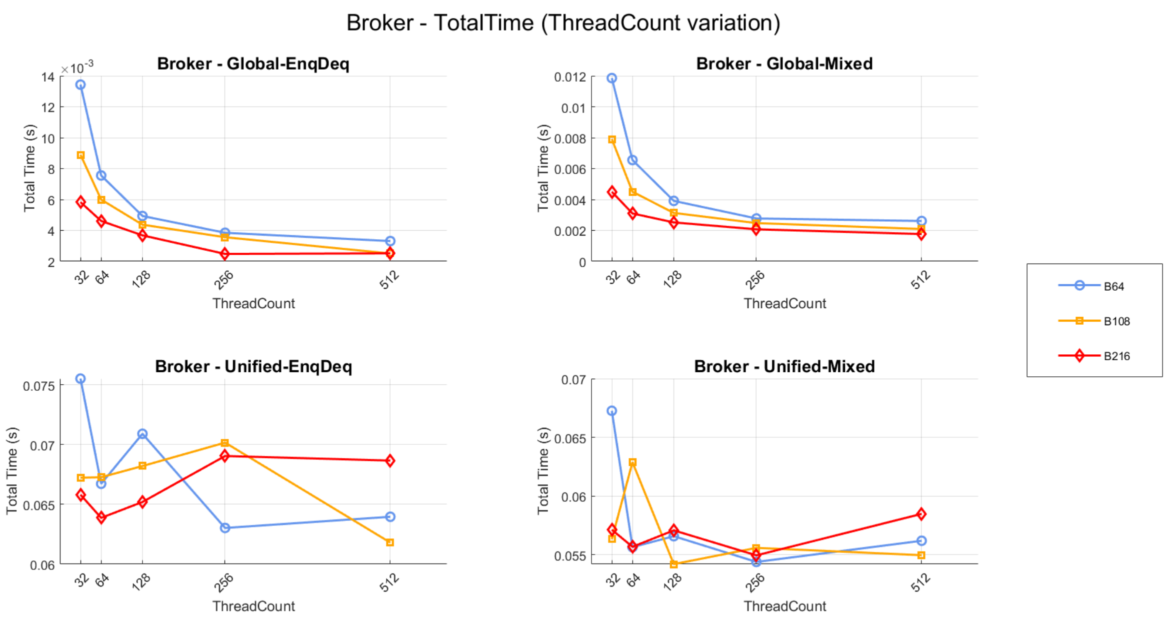
Broker Queue Performance

As illustrated in Figure 4.1a and 4.1b, the results, both the Global-Enqueue-Dequeue (EnqDeq) and Global-Mixed scenarios show that the Broker achieves a significant reduction in total execution time as the thread count increases, particularly between 32 and 256 threads. This trend indicates that the Broker efficiently manages increased concurrency, likely due to its design that allows multiple threads to perform enqueue and dequeue operations simultaneously. It is evident that as both thread and block counts rise, the total execution time decreases, which is expected since the workload is distributed among more threads. However, when the thread count exceeds 256, the performance gains plateau, suggesting that the system may be approaching its optimal capacity for handling concurrent operations. Also it can be observed that when thread and block counts are increasing, in Unified Memory the performance is not improving as stable as it was in Global Memory. This is an expected result since NVIDIA is working on performance improvement for Unified Memory since its generally slower than Global Memory [NVIDIA, 2024].

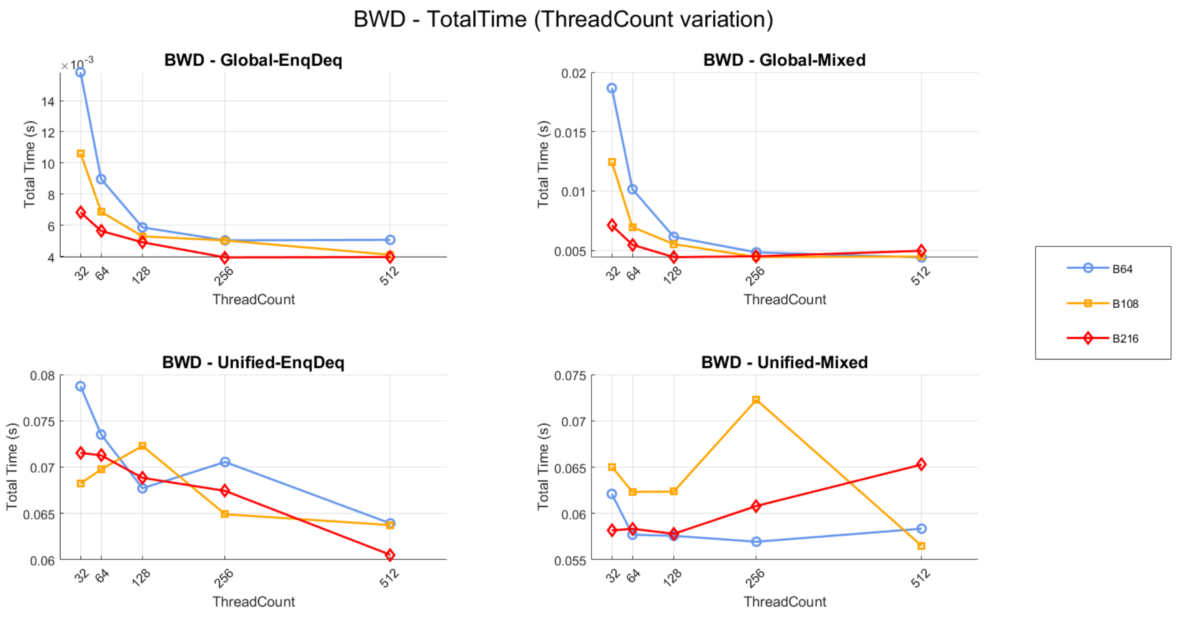
In the Global-Mixed scenario, the Broker maintains a similar trend, with total execution time decreasing as the thread count increases. The results demonstrate that the Broker effectively balances enqueue and dequeue operations, which is crucial for sustaining throughput in mixed workloads. The consistent performance across varying thread counts reinforces the Broker's robustness in managing diverse operational demands. Notably, when comparing unified memory to global memory, the Broker shows improved performance with global memory, as it allows for more efficient data access patterns and reduced latency, particularly in high-concurrency scenarios.

Broker Work Distributor Queue Performance

In contrast, the BWD exhibits a different performance profile. In the Unified-EnqDeq scenario, the total execution time fluctuates more significantly across thread counts. This inconsistency suggests that the BWD may struggle to maintain efficiency under varying loads, potentially due to its non-linearizability. The BWD's reliance on a single call to ensureEnqueue and ensureDequeue can lead to situations where the queue state is not accurately represented, resulting in increased execution times when the queue is under heavy contention. When using unified memory, the BWD benefits from the shared memory space between the CPU and GPU, which eliminates the need for data transfers and allows for more straightforward access patterns. This can enhance ease of programming and reduce complexity. In contrast, global memory often demonstrates better performance in high-concurrency scenarios, as it allows for more efficient data access patterns and reduced latency. It can be interpreted from figures that Broker Queue is particularly effective in managing workloads that require rapid enqueue and dequeue operations, further emphasizing the importance of selecting the appropriate memory model based on the specific performance needs of the application.



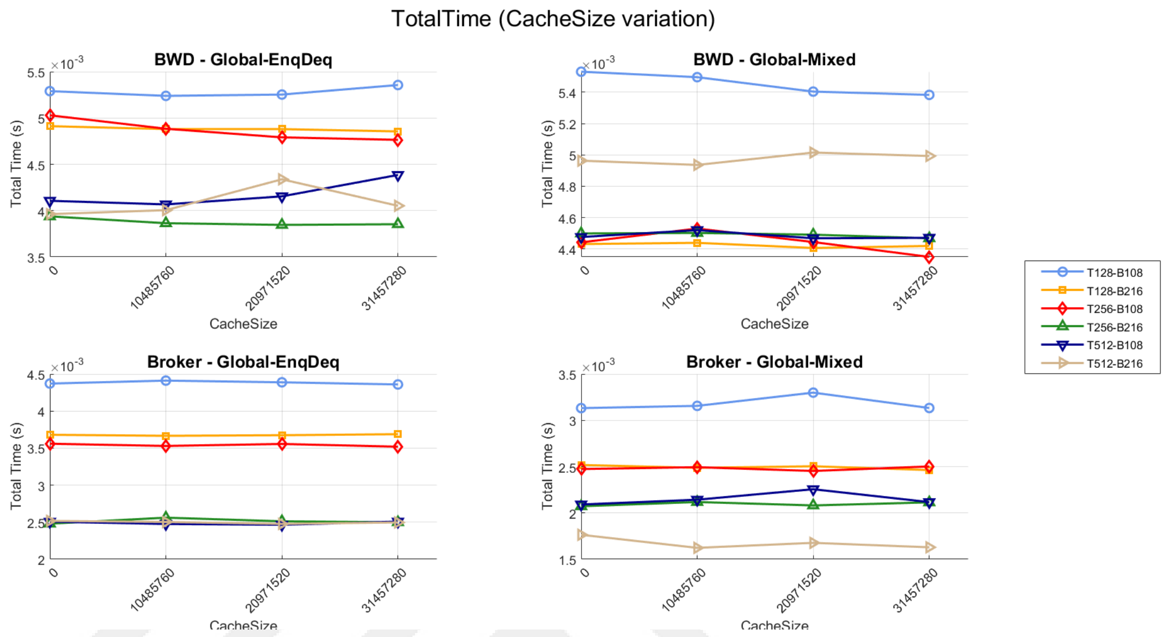
(a)



(b)

Figure 4.1: (a) Single-GPU Broker Queue Unified and Global Memory under different thread and block counts, (b) Single-GPU BWD Queue Unified and Global Memory under different thread and block counts

4.1.2 L2 Cache Set-Aside Variations



(a)

Figure 4.2: Single-GPU Broker and BWD Queues Cache Size Variations under different thread and block counts

In examining the variations in L2 Cache Set-Aside in Figure 4.2a and 4.2b, it is evident that there is minimal impact on performance for both the Broker and Broker Work Distributor (BWD) queues across the Global-Enqueue-Dequeue (EnqDeq) and Global-Mixed scenarios. This lack of significant performance improvement may be attributed to several factors.

Firstly, the inherent design of both queue implementations may not fully exploit the benefits of L2 cache optimizations, leading to limited gains from increased cache allocation. Additionally, the workloads tested may not have been sufficiently cache-sensitive, meaning that the data access patterns did not benefit from the additional cache space. Furthermore, the overhead associated with managing cache allocations could offset any potential performance gains, resulting in a relatively stable execution time across different cache sizes. Additionally, given that the experiments involve at most 110,592 (512 threads * 256 blocks) threads performing 10 million enqueue operations, the element array being pushed into the queue may not have been accessed persistently or frequently enough to benefit from L2 cache set-aside. This lack of persistent or localized data access could result in the underutilization of the allocated cache space.

This suggests that while cache management is important, the specific characteristics of the workload and the architecture of the queue implementations play a crucial role in determining overall performance. The limited reliance on repeated or frequent data access patterns could explain why the L2 cache optimization does not significantly impact performance.

4.2 Multi-GPU Queue Implementation and Speedup Results

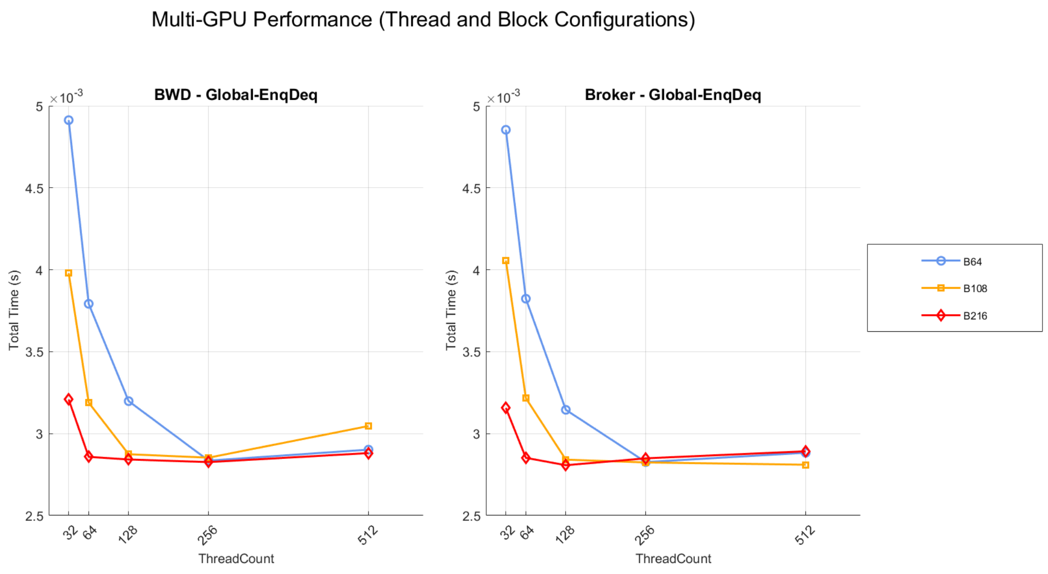


Figure 4.3: Multi-GPU Broker and BWD Queues under different thread and block counts

Figure 4.3 illustrates the multi-GPU performance of both the Broker and Broker Work Distributor (BWD) in the Global-Enqueue-Dequeue (EnqDeq) scenario, focusing on various thread and block configurations. For the both queue implementations, the total execution time shows a noticeable decrease as the both thread and block counts increase, particularly between 32 and 256 threads and for 216 blocks. This trend indicates that both queues can effectively utilize additional threads to enhance performance, likely due to improved parallelism in enqueue and dequeue operations. However, as the thread count continues to rise beyond 256, the performance gains begin to plateau, suggesting that the queues may be reaching its optimal capacity for handling concurrent operations.

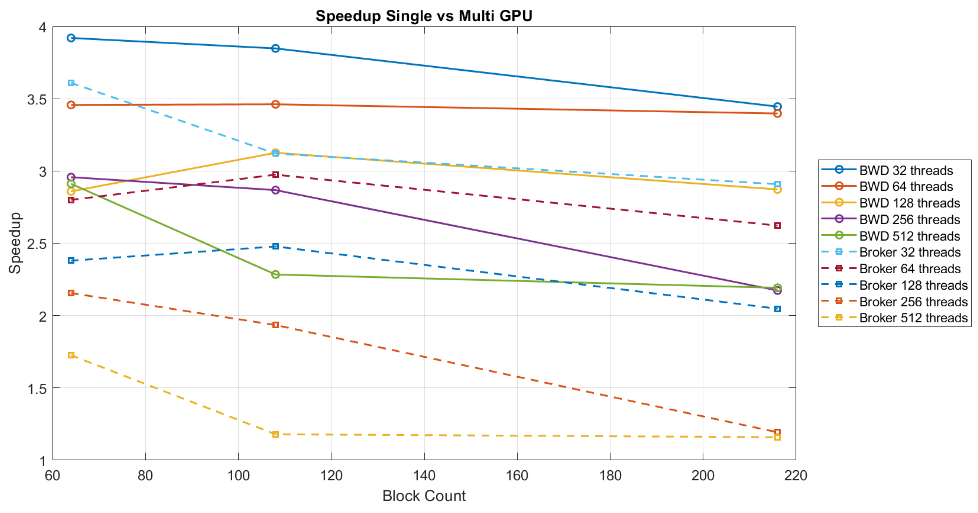


Figure 4.4: Speedup Single-GPU vs Multi-GPU Broker and BWD Queues under different thread and block counts

Figure 4.4 illustrates the speedup achieved by both the Broker and BWD queues when transitioning from single-GPU to multi-GPU configurations across various block counts. Both implementations exhibit similar trends: as the thread count increases, the speedup becomes less pronounced. This behavior suggests that, while both queues benefit from multi-GPU setups, the efficiency gains diminish at higher thread counts. One possible explanation for this trend is that the single-GPU configuration operates very efficiently with larger thread counts, effectively utilizing the available resources. In contrast, the overhead associated with managing multiple GPUs may introduce latency, which can offset the expected performance improvements. The maximum speedup for Broker Queue is 3.92 and the average Speedup is 3.04, for the BWD Queue the maximum speedup is 3.88 and the average is 3.02. Additionally, tests conducted with 2 GPUs shows a maximum speedup of 2.425 and an average speedup of 2.168. This indicates that performance may degrade when the GPU count is lower, as the workload is distributed across fewer GPUs, which can lead to reduced parallelism and less effective resource utilization. These results are significantly proving that taking the queue implementation to multi-GPU as a novel work of this thesis makes a great improvement.

4.3 Bellman Ford Implementation Results

4.3.1 Single-GPU Bellman Ford

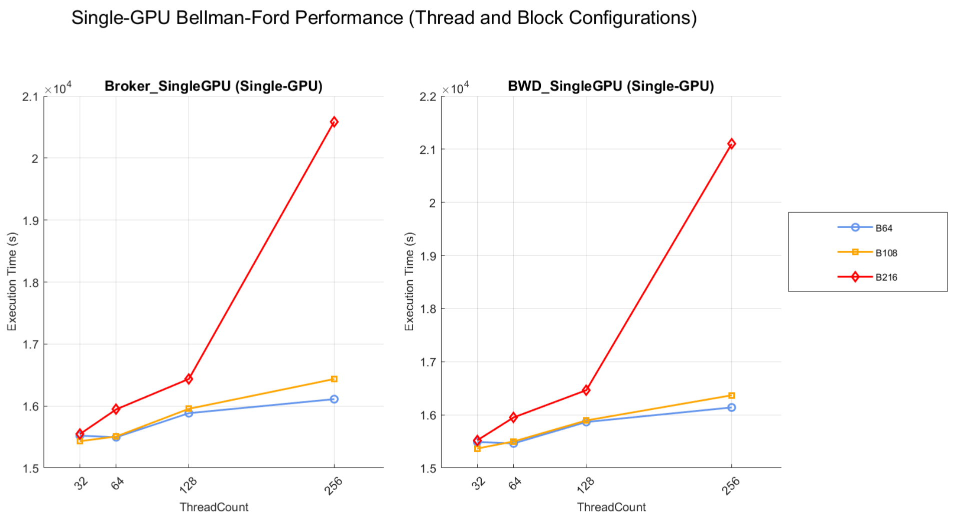


Figure 4.5: Single-GPU BF for Broker and BWD Queues under different thread and block counts

The figure 4.5 illustrates the execution time of the Bellman-Ford algorithm implemented using both the Broker and Broker Work Distributor (BWD) queues on a single GPU, with varying thread counts and block configurations. Both implementations exhibit a similar trend in execution time as the thread count increases. Notably, execution times rise gradually, particularly at higher thread counts (128 and 256). The performance across different block sizes (64, 108, 216) remains relatively close, with 64 consistently demonstrating the lowest execution time across all thread counts. This suggests that both queue implementations effectively manage the workload, with smaller block sizes potentially allowing for better resource utilization and reduced overhead [NVIDIA, 2024]. However, as the thread count increases, the execution time for both implementations tends to rise, indicating that while they can handle the workload, the efficiency gains diminish at higher thread counts. This behavior may be attributed to several factors. Higher thread counts can lead to lower occupancy if the number of threads per block exceeds the optimal range for the GPU architecture, resulting in underutilization of GPU resources. Additionally, increasing the number of threads can lead to increased memory access, which may saturate the memory bandwidth. If the algorithm becomes memory-bound, this can lead to performance degradation. Moreover, the overhead associated with launching kernels can become significant when the number of threads is very high relative to the number of blocks, offsetting the benefits of parallelism. As the number of threads increases, contention for shared resources, particularly in atomic operations, can lead to serialization, negatively impacting performance. Furthermore, the Bellman-Ford algorithm has inherent dependencies that may limit the scalability of parallelism. As more threads are added, the workload may not be evenly distributed, leading to idle time for some threads. In summary, while both queue implementations demonstrate the ability to manage the workload effectively, the observed increase in execution time at higher thread counts suggests that the overhead associated with managing more threads can offset the benefits of parallelism. This highlights the importance of optimizing thread and block configurations to achieve the best performance.

4.3.2 Multi-GPU Bellman Ford and Speedup

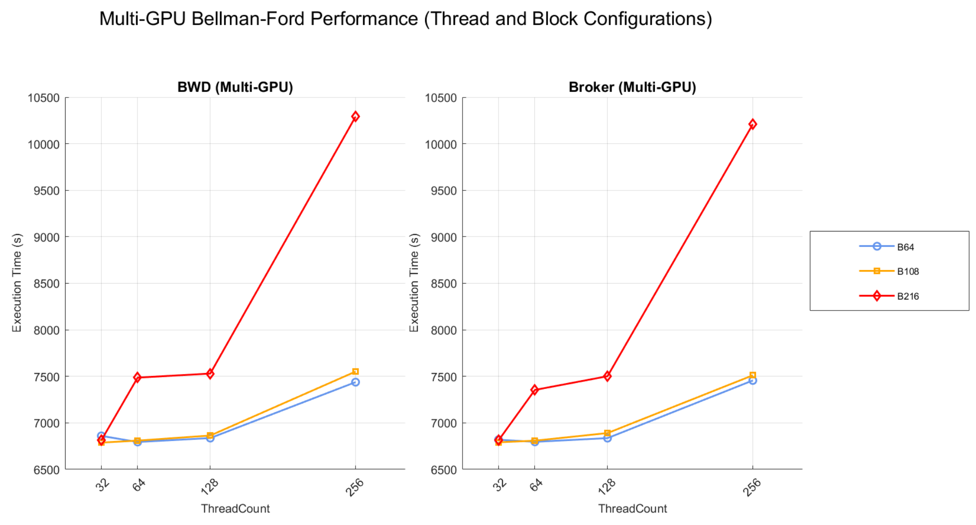
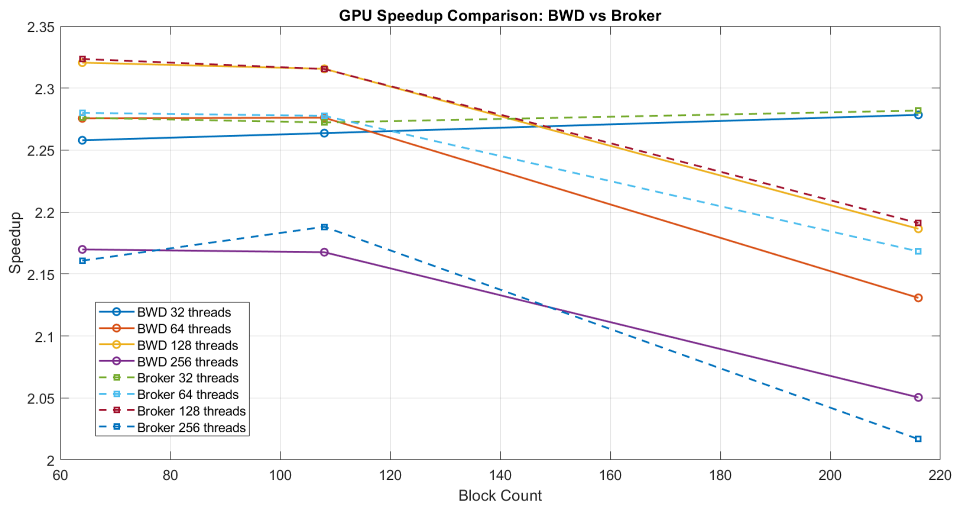


Figure 4.6: Multi-GPU BF for Broker and BWD Queues under different thread and block counts

The figure 4.6 illustrates the execution time of the Bellman-Ford algorithm implemented using both the Broker and Broker Work Distributor (BWD) queues in a multi-GPU environment, with varying thread counts and block configurations. Both implementations exhibit a similar performance trend, with execution times generally increasing as the thread count rises. At lower thread counts (32 and 64), the execution times are relatively low, indicating that both queue designs can efficiently manage the workload. However, as the thread count increases to 128 and 256, the execution times rise significantly, suggesting that the overhead associated with managing multiple GPUs may be impacting performance. This behavior highlights a potential bottleneck, where the benefits of parallelism are offset by the complexities of coordinating tasks across multiple GPUs.

As seen in Figure 4.6, although 216 blocks performed worse, across different block configurations (64, 108, 216), the execution times remain fairly consistent, with minimal variation observed. This indicates that neither implementation significantly benefits from changing block sizes in this context, suggesting that the workload characteristics may not be sensitive to block size adjustments. Overall, the analysis reveals that while both the Broker and BWD can handle the Bellman-Ford algorithm in a multi-GPU setup, the increasing execution times at higher thread counts point to challenges in scaling performance effectively. This underscores the importance of optimizing both the algorithm and the queue design to fully leverage the capabilities of multi-GPU environments, ensuring that the overhead of coordination does not negate the advantages of parallel processing.



(a)

Figure 4.7: Speedup BF of Single-GPU vs Multi-GPU Broker and BWD Queues under different thread and block counts

The speedup graph compares the performance of the Broker and BWD queue implementations for the Bellman-Ford algorithm in both single-GPU and multi-GPU environments, highlighting key trends across different thread counts and block configurations. Both implementations exhibit a gradual decline in speedup as the block count increases, indicating that while multi-GPU setups provide some performance benefits and shows better performance for 64 and 108 blocks, it may be effected by the increasing overheads and gets worse for 216 blocks. For Broker queue, the maximum speedup is 2.315 and the average is 2.229. For BWD queue, the maximum speedup is 2.325 and the average is 2.224. This analysis demonstrates that both queue implementations exhibit a significant performance improvement in multi-GPU configurations compared to single-GPU setups, as anticipated. While Section 4.2 and Figure 4.4 indicates that speedup ratios for queue implementations could reach up to 3.92x, the Bellman-Ford application may encounter additional overhead from other components of the algorithm in the multi-GPU version. Nevertheless, achieving a speedup of 2.32x remains a commendable outcome, highlighting the effectiveness of both queue implementations in leveraging multi-GPU resources.

Graph	**Kind**	**Vertex Count**	**Edge Count**	**Maximum Speedup**	**Average Speedup**
G65	Random Matrix	8000	8000	3.704	3.485
pcb3000	Linear Programming	3960	7732	3.794	3.573
G67	Random Matrix	10000	10000	3.183	3.011
lp_ken_11	Linear Programming	14694	21349	3.103	2.964
ch7-6-b5	Combinatorial Problem	5040	15120	2.997	2.454
Reuters911	Network	13332	13332	2.325	2.239
Foldoc	Network,Computing	13356	13356	2.323	2.237
G39	Random Matrix	2000	2000	2.294	2.140
Cyl6	Structural Problem	13681	13681	2.319	2.238
Alemdar	2D/3D Problem	6245	6245	2.328	2.219

Table 4.1: SuiteSparse Graphs [SuiteSparse, 2024] and Single-GPU vs. Multi-GPU Speedup Results using 4 GPUs

Furthermore, as illustrated by SuiteSparse graphs in Table 4.1, the results indicate that higher speedups were achieved compared to those observed with the artificially generated benchmark discussed earlier. This discrepancy may be attributed to the inherent characteristics of the graphs used in the SuiteSparse collection, which often exhibit more complex connectivity patterns and structures. Such connectivity can lead to more efficient data access patterns and reduced contention during the execution of the Bellman-Ford algorithm. The diverse nature of the graphs, including variations in vertex and edge counts, likely contributes to the improved performance, as they may better exploit the parallelism offered by multi-GPU configurations. This suggests that the choice of graph structure plays a crucial role in determining the effectiveness of parallel algorithms, emphasizing the importance of using representative datasets for performance evaluation. Notably, the best graph from the SuiteSparse collection, pcb3000 which is a linear programming problem, achieved a maximum speedup of 3.794x and an average speedup of 3.573x, underscoring the potential for significant performance gains when utilizing well-structured graph data.

Chapter 5

CONCLUSION

This thesis presents a pioneering implementation of a multi-GPU concurrent queue system, establishing a significant advancement in high-performance computing. The proposed multi-GPU queue is the first of its kind in the literature, extending the capabilities of existing single-GPU queue structures to a multi-GPU environment. This innovation addresses critical challenges related to scalability and efficiency in parallel processing, particularly in the context of concurrent data structures.

The performance evaluations conducted demonstrate that the multi-GPU queue achieves remarkable speedup ratios, with a maximum speedup of 3.92x and an average speedup of 3.04x across various configurations on four NVIDIA A100 GPUs. These results highlight the effectiveness of the multi-GPU approach in leveraging the computational power of multiple GPUs, significantly enhancing throughput and reducing latency in queue operations. Such performance improvements are crucial for applications requiring high concurrency and rapid graph data processing.

Additionally, the application of the Bellman-Ford algorithm tested by utilizing this novel multi-GPU queue represents a novel contribution to the field, as no prior research has explored the optimization of this algorithm using concurrent queue systems in a multi-GPU context. The multi-GPU Bellman-Ford implementation achieved a maximum speedup of 3.794 and an average speedup of 3.573, demonstrating its capability to efficiently handle large-scale graph processing tasks. This implementation not only showcases the potential of multi-GPU systems for solving complex graph algorithms but also emphasizes the importance of effective data management in parallel computing environments. As a future work, this thesis can be extended to the other SSSP algorithms and it can be tested with 8 or more GPUs to see if the speedup is increasing.

In conclusion, this research significantly advances the state-of-the-art in concurrent queue applications and multi-GPU implementations. By addressing both theoretical and practical challenges, this thesis lays a robust foundation for future exploration of multi-GPU systems across various computational domains. The findings underscore the potential for further research to enhance the performance and applicability of high-performance computing technologies, ultimately contributing to the ongoing evolution of parallel processing methodologies.

BIBLIOGRAPHY

[Agarwal and Dutta, 2015] Agarwal, P. and Dutta, M. (2015). New approach of bellman ford algorithm on gpu using compute unified design architecture (cuda). *International Journal of Computer Applications*, 110(13):975–8887.

[Anthony Williams, 2024] Anthony Williams (2015 (accessed December 10, 2024)). Graduation checklist. `https://www.justsoftwaresolutions.co.uk/files/safety_off.pdf`.

[Barnes, 1993] Barnes, G. (1993). A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270.

[Bellman, 1958] Bellman, R. (1958). On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90.

[Busato and Bombieri, 2015] Busato, F. and Bombieri, N. (2015). An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233.

[Cederman et al., 2012] Cederman, D., Chatterjee, B., and Tsigas, P. (2012). Understanding the performance of concurrent data structures on graphics processors. In *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*, pages 883–894. Springer.

[Chakaravarthy et al., 2016] Chakaravarthy, V. T., Checconi, F., Murali, P., Petrini, F., and Sabharwal, Y. (2016). Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2031–2045.

[Chen et al., 2007] Chen, M., Chowdhury, R. A., Ramachandran, V., Roche, D. L., and Tong, L. (2007). Priority queues and dijkstra's algorithm.

[Cormen et al., 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to algorithms*. MIT press.

[Ford, 1956] Ford, L. R. (1956). Network flow theory. *Rand Corporation Paper, Santa Monica, 1956*.

[Gottlieb et al., 1983] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. (1983). Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):164–189.

[Haas et al., 2012] Haas, A., Kirsch, C. M., Lippautz, M., and Payer, H. (2012). How fifo is your concurrent fifo queue? In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 1–8.

[Herlihy, 1993] Herlihy, M. (1993). A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770.

[Herlihy et al., 2003] Herlihy, M., Luchangco, V., and Moir, M. (2003). Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE.

[Hsu et al., 2020] Hsu, C.-H., Imam, N., Langer, A., Potluri, S., and Newburn, C. J. (2020). An initial assessment of nvshmem for high performance computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–10. IEEE.

[Hwang and Faye, 1984] Hwang, K. and Faye, A. (1984). Computer architecture and parallel processing.

[Kerbl et al., 2018] Kerbl, B., Kenzel, M., Mueller, J. H., Schmalstieg, D., and Steinberger, M. (2018). The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 76–85.

[Madkour et al., 2017] Madkour, A., Aref, W. G., Rehman, F. U., Rahman, M. A., and Basalamah, S. (2017). A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*.

[Mellor-Crummey, 1987] Mellor-Crummey, J. (1987). Concurrent queues: Practical fetch-and-φ algorithms.

[Michael and Scott, 1996] Michael, M. M. and 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*, pages 267–275.

[Misra and Chaudhuri, 2012] Misra, P. and Chaudhuri, M. (2012). Performance evaluation of concurrent lock-free data structures on gpus. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 53–60. IEEE.

[Moore, 1959] Moore, E. F. (1959). The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press.

[MPI, 2024] MPI (2023 (accessed December 9, 2024)). Mpi guide. `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf/`.

[NVIDIA, 2024] NVIDIA (2024 (accessed December 9, 2024)). Cuda c programming guide. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf/`.

[NVIDIA2, 2024] NVIDIA2 (2024 (accessed December 9, 2024)). Nvshmem guide. `https://docs.nvidia.com/nvshmem/api/index.html/`.

[Prakash et al., 1991] Prakash, S., Lee, Y.-H., and Johnson, T. (1991). Non-blocking algorithms for concurrent data structures. Master's thesis, Citeseer.

[Prakash et al., 1994] Prakash, S., Lee, Y. H., and Johnson, T. (1994). A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559.

[Sites, 1978] Sites, R. (1978). *Operating systems and computer architecture*.

[Stone, 1990a] Stone, H. S. (1990a). *High-performance computer architecture*. Addison-Wesley Longman Publishing Co., Inc.

[Stone, 1990b] Stone, J. M. (1990b). A simple and correct shared-queue algorithm using compare-and-swap. In *Supercomputing'90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 495–504. IEEE.

[Stone, 1992] Stone, J. M. (1992). A non-blocking compare-and-swap algorithm for a shared circular queue. *S. Tzafestas et al., editors, Parallel and Distributed Computing in Engineering Systems*, pages 147–152.

[SuiteSparse, 2024] SuiteSparse (2024 (accessed December 9, 2024)). Suitesparse matrix collection. `https://sparse.tamu.edu/`.

[Zhang et al., 2014] Zhang, X., Deng, Y., and Mu, S. (2014). Toward concurrent lock-free queues on gpus. *IEICE TRANSACTIONS on Information and Systems*, 97(7):1901–1904.

[Zhang et al., 2020] Zhang, Y., Brahmakshatriya, A., Chen, X., Dhulipala, L., Kamil, S., Amarasinghe, S., and Shun, J. (2020). Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 158–170.