Fully GPU-Orchestrated Multi-GPU Work Stealing

Master's Thesis
of

Lennard Kittner

at the Department of Informatics
ITEC - Operating Systems Group

First Reviewer:	Prof. Dr. Frank Bellosa
Advisor:	Peter Maucher

Completion period:	16. July 2025 - 16. January 2026

Statutory Declaration

I hereby declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, January 16, 2025

....

(Lennard Kittner)

Abstract

Since the introduction of general-purpose GPU compute (GPGPU), GPUs have become an essential part of high-performance and scientific computing. However, efficiently utilizing the vast compute resources, especially in multi-GPU environments with generic irregular workloads, necessitates load balancing. Existing approaches typically rely on the CPU to manage work on behalf of the GPU.

In this thesis, we propose MGWS, a novel decentralized work stealing system that allows GPU workers to operate independently of the CPU. Removing the reliance on the CPU prevents CPU-managed threads from becoming a bottleneck and reduces synchronization and communication overhead between the CPU and GPU.

We present two inter-GPU communication schemes: the first is based on CPU host memory mapped to all GPUs, and the second leverages peer-to-peer direct memory access (DMA) for direct GPU-to-GPU communication. Our experiments demonstrate that the choice of inter-GPU communication mechanism has a substantial impact on overall performance. Due to hardware limitations, the full multi-GPU evaluation is restricted to the host memory-based communication scheme; however, preliminary tests suggest that using host memory is likely slower than peer-to-peer DMA. Even with only host memory enabled, MGWS can still outperform a static task assignment by up to 41%.

Zusammenfassung

Seit der Einführung von General-Purpose-GPU-Compute (GPGPU) sind GPUs zu einem essenziellen Bestandteil des High-Performance- und Scientific-Computings geworden. Um diese immensen Rechenkapazitäten effizient zu nutzen, insbesondere in Multi-GPU-Umgebungen mit generischen, irregulären Workloads, ist jedoch eine Lastverteilung erforderlich, die bei bestehenden Systemen typischerweise von der CPU für die GPU übernommen wird.

In dieser Arbeit, stellen wir MGWS vor, ein neuartiges dezentralisiertes Work-Stealing-System, das es GPU-Arbeitern ermöglicht, unabhängig von der CPU zu agieren. Durch das Entfernen der Abhängigkeit von der CPU wird verhindert, dass CPU-Threads zum Engpass werden, während gleichzeitig der durch Synchronisation und Kommunikation zwischen CPU und GPU entstehende Overhead reduziert wird.

Wir präsentieren zwei Inter-GPU-Kommunikationsmechanismen: Der erste beruht auf dem Hauptspeicher, der auf allen GPUs gemappt wird, während der zweite Peer-to-Peer Direct Memory Access (DMA) für direkte GPU-zu-GPU-Kommunikation nutzt. Unsere Experimente zeigen, dass die Wahl des inter-GPU-Kommunikationsmechanismus einen erheblichen Einfluss auf die Gesamtperformanz hat. Aufgrund von Hardwareeinschränkungen beschränkt sich die Multi-GPU-Evaluation auf den hauptspeicherbasierten Ansatz. Allerdings zeigen vorläufige Testergebnisse, dass das Nutzen von Hauptspeicher vermutlich langsamer ist als der Peer-to-Peer-DMA-Ansatz. Dennoch kann MGWS eine statische Lastverteilung für alle bis auf die höchste Anzahl an Workern um bis zu 41,09% übertreffen.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 3

2. Background 7

2.1. GPU Architecture 7
2.2. HIP (Heterogeneous-Compute Interface for Portability) 8
2.3. Persistent Kernels 9
2.4. Peer-to-Peer Communication 10
2.5. Challenges Utilizing Multiple GPUs 10

3. Related Work 11

3.1. Load Balancing 11
3.1.1. Work Stealing 11
3.1.2. Structured-Atomic Work Stealing System (SWS) 12
3.2. GPU Load Balancing 13
3.2.1. Task-Specific GPU Work Stealing 13
3.2.2. Single-GPU Load Balancing 14
3.2.3. Multi-GPU Work Stealing 16

4. Design 21

4.1. Persistent Kernels 21
4.2. Work Stealing 21
4.3. Steal Counter Overflow Prevention 23
4.4. Differences to SWS 25
4.5. Task Struct 27
4.6. Termination Detection 28
4.7. Multi-GPU 28
4.8. Data Management 30

5. Implementation 31

5.1. Work Stealing Algorithm 31
5.2. Task Struct 33
5.3. Orchestration Data 36
5.4. Persistent Kernel 37
5.5. Target Selection 38
5.6. Termination Detection 40
5.6.1. Atomic Counter Termination Detection 41
5.6.2. Tree-Based Termination Detection 41
5.6.3. Race Conditions 42

5.7. Determine the Number of Workers for Maximum Occupancy 45
5.8. Cross-Platform Work Stealing 46
5.9. Utilizing Peer-to-Peer DMA 47

6. Evaluation 49
6.1. Test Systems 50
6.2. Workloads 50
6.3. Correctness 51
6.4. Effects of Fully Utilizing the GPU with Persistent Kernels 53
6.5. Overhead of Overflow Prevention 54
6.6. Termination Detection Mechanisms 56
6.7. Multiple Queue Owners 58
6.8. Single-GPU Evaluation 62
6.8.1. Comparison Between RRX6950 and RX6600 62
6.8.2. RX6950 Regular Workload 63
6.8.3. RX6950 Irregular Workload 66
6.9. Multi-Device Evaluation 71
6.9.1. Multi-Device Work Stealing with a Regular Workload 72
6.9.2. Multi-Device Work Stealing with an Irregular Workload 76
6.10. Understanding the Performance Impact of Host Memory-Based Communication 80
6.11. Discussion 81

7. Future Work 87

8. Conclusion 91

Bibliography 93

Appendix 97
A Memory Latency Measurement Code 97
B Test Stream Destruction behavior 99
C Test Peer-To-Peer Atomics 101
D Replicate Evaluation 103
D Create Wikipedia Data 103
D Build MGWS 103
D Reproduce The Results 103
D Display The Results 103

1. Introduction

Graphics Processing Units (GPUs) play a crucial role in modern high-performance computing (HPC), particularly in the fields of artificial intelligence (AI) [26], but also physics [30,38,48] and graph applications [51,52,64]. According to the November 2024 TOP500 [1] list, nine out of the ten fastest supercomputers incorporate GPUs.

Massively parallel workloads, which break large problems down into many small, simple, independent tasks using strategies such as MapReduce, are typically used to take full advantage of the computational power of GPUs [39]. Many applications can also utilize multiple GPUs simultaneously. Irregular workloads feature tasks where the execution times are not known in advance; examples are graph traversal [52], the boolean satisfiability problem (SAT) [34], and branch-and-bound [33]. These workloads profit from load balancing to utilize GPU resources efficiently. In such scenarios, work stealing is a capable load balancing technique [22].

An effective load balancing algorithm should distribute work evenly across GPU SIMD processors and minimize overhead so that more compute time can be spent completing the actual task. However, many existing GPU load balancing solutions rely on the CPU for mapping tasks to GPU workers, which introduces latency due to the need for GPU-CPU synchronization and communication [20,31,32,42,46].

Dynamic load balancing strategies such as work stealing allow tasks to spawn new subtasks, which often access the same data as their parent; thus, scheduling the subtask on the same processor as the parent usually leads to better cache utilization [24]. Some systems, however, represent tasks as GPU kernels; while this gives the programmer a familiar interface, it has some drawbacks [20,32,46], mainly reliance on the hardware scheduler and that kernels must run to completion and cannot be suspended or migrated. Thus, if a GPU's compute units are fully occupied, newly spawned kernels must wait, delaying the execution of new kernels and subtasks that are less likely to be scheduled on the same processor as their parent.

While there are GPU-based load balancing mechanisms that eliminate CPU involvement [24,25,51,65,66], they are typically limited to specific workloads or single-GPU environments. To our knowledge, no general-purpose multi-GPU work stealing system exists that is not managed by the CPU.

To address this gap, we propose *Multi GPU Work Stealing (MGWS)*, a fully GPU-orchestrated, multi-GPU work stealing system not limited to specific workloads. Therefore, the CPU is limited to participating as a worker and does not manage task distribution for GPU workers. Our approach reduces CPU communication and synchronization overheads by utilizing a decentralized structure, where GPU workers can execute and steal tasks without CPU involvement. It can further eliminate host memory accesses after the initial task loading by leveraging peer-to-peer DMA for direct communication between GPUs. Given proper hardware support, MGWS can theoretically support CPU workers and GPUs from different vendors at the same time. However, we were unable to test work stealing across GPU vendors because the NVIDIA GPUs we had access to lacked proper PCIe atomics support. We are also not limited to a specific workload and can handle custom user-defined irregular tasks.

Our implementation is based upon the Structured-atomic Work Stealing (SWS) algorithm by Cartier et al. [23], which utilizes separate queues for stealable and non-stealable tasks, which suits the hierarchical memory architecture of modern GPUs well. We propose two configurations: one uses peer-to-peer Direct Memory Access (DMA) for direct inter-GPU communication without CPU involvement, while the other employs DMA on registered (i.e., usually page-locked and GPU-mapped) host memory for environments where peer-to-peer DMA is unavailable. Our evaluation system supports peer-to-peer DMA, but atomics over this peer-to-peer connection are not coherent, which is necessary for MGWS. We therefore did not evaluate the peer-to-peer DMA communication scheme directly but approximated its performance by storing and accessing data located on a remote GPU in a single GPU setting. Inter-GPU data migration is out of scope for this thesis, but when designing MGWS, we already included the possibility for future active data migrations by including information about the data a task relies on in the task struct.

In our evaluation, we analyze all major parts of MGWS, including various design alternatives. To that end, we are utilizing two workloads. The first uses short, regular tasks that only write a given value to memory called *MEMSET*. The second workload is irregular, where each task determines whether a Wikipedia article contains a given word. We call this workload *CONTAINS*. We employ the MEMSET workload to assess the overhead caused by MGWS and the CONTAINS workload to observe how MGWS behaves when load balancing irregular tasks that dominate the overall runtime.

In a single GPU environment under the MEMSET workload, we observe substantial performance differences between different inter-worker communication strategies; using host memory is up to 246 times slower compared to global memory. We discuss reasons for this large performance gap in Section 6.10. Using the irregular workload, this is less pronounced, where using host memory is faster than global memory at low worker counts and only 2.28 times slower than global memory at high worker counts.

Since our evaluation system does not support coherent atomics over the peer-to-peer connection between the GPUs, we were unable to directly evaluate our peer-to-peer DMA-based communication scheme. However, we were able to place task queues and other MGWS orchestration data in the memory of a remote GPU, allowing us to approximate the performance achievable with the proper peer-to-peer DMA communication scheme. In most experiments, remote global memory performs similarly to local global memory. Based on this, we expect that the peer-to-peer DMA-based inter-GPU communication scheme will not suffer the same significant performance loss sometimes observed using the host memory-based communication approach.

We also compare MGWS with direct task execution using both workloads with a static task assignment. Since our hardware restricts MGWS to the host memory-based inter-GPU communication approach, we observe a significant performance difference of up to 129 times under the regular workload with multiple GPUs. However, under the irregular workload or CPU worker participation, MGWS outperforms direct task execution for all but the highest worker counts.

This thesis is divided into three parts. The first part, containing Chapter 2 and 3, provides background on GPU architecture, which we are going to exploit to improve the performance of our algorithm. Furthermore, it encompasses prior research on GPU load balancing, which involves categorizing them according to their multi-GPU support, CPU involvement, and support for general-purpose tasks. Chapter 4 and 5 form the second part: here we first discuss design decisions we made during the planning of MGWS and also highlight some alternative designs we considered. This is followed by a description of our implementation, where we give a more detailed view of MGWS, including how we manage to facilitate stealing between NVIDIA and AMD GPUs. The remaining chapters form the third part, starting with the evaluation in Chapter 6, which compares various configurations of MGWS and analyzes the bottleneck caused by relying on host memory for inter-GPU and inter-worker communication. In Chapter 7, we motivate further extensions of MGWS, including approaches to reduce the overhead caused by host memory-based communication. Finally, we conclude our thesis with a summary of our approach and findings in Chapter 8.

2. Background

This section provides insights into GPU architecture, with a particular focus on the memory hierarchy (Section 2.1). Next, we present ROCm HIP [4] a framework developed by AMD for writing GPU kernels and managing device execution (Section 2.2). Afterwards we discuss persistent GPU kernels, which allow MGWS to reduce its dependency on the CPU (Section 2.3). Then, we give a brief overview of peer-to-peer communication, which enables direct communication between GPUs (Section 2.4). Finally, we list several challenges that may occur when utilizing multiple GPUs simultaneously (Section 2.5).

2.1. GPU Architecture

Historically, the GPU's primary purpose was graphics processing; therefore, its architecture is designed with that goal in mind. However, nowadays, GPUs are increasingly used for general-purpose computing as well, using APIs such as HIP [4], CUDA [54], OpenCL [35], and Vulkan [36].

A GPU is a multithreaded single instruction multiple data (SIMD) processor [37]. Thus, it consists of multiple processing cores (*SIMD processors*), where each core operates on multiple pieces of data simultaneously. AMD refers to these SIMD processors as *compute units* (CU) [5]. The part of the SIMD processor that operates on a single piece of data is called a *SIMD lane*.

The memory hierarchy is an important part of the GPU architecture. It consists of *global memory* (also known as *device memory*), *local memory*, *shared memory*, and *registers*. The global memory, typically several gigabytes in size and DRAM-based, is comparatively slow with access latencies between 200 and 1000 cycles, but it can be accessed by all SIMD lanes [37,40,53]. Each SIMD lane also has its own private section of off-chip DRAM called local memory, which is used in case of register spillover [37,55]. In addition to local memory, there is a small amount of SRAM-based memory partitioned into shared memory and L1 cache [40,53], with access latencies as low as 20-30 cycles. This memory is significantly smaller than the global memory, typically limited to tens to hundreds of kilobytes per CU, and cannot be shared between SIMD processors [37,40,53]. A GPU also contains an L2 cache, which is shared across all CUs, with access latencies of approximately 200 cycles and a size of around 4 MB [6,40,53]. Recent AMD GPUs also include a so-called *AMD Infinity Cache™*, a large 256 MB on-chip L3 cache shared by all CUs [6]. Since the introduction of the infinity cache, one of the L2 cache's main purposes is to coalesce memory accesses [6]. Finally, each SIMD lane has a dedicated register file, likewise sized in the range of tens to hundreds of kilobytes per CU [37,40,53].

GPUs try to minimize the number of individual global memory accesses by coalescing accesses from multiple SIMD lanes [56]. Each global memory transaction is 32 B; by letting consecutive SIMD lanes access consecutive locations in global memory, a single transaction can supply multiple lanes with data [56]. Thus, reducing the total number of global memory accesses.

The GPU can also perform DMA on host memory if it is *registered*, which entails mapping the memory to the GPU and usually also page-locking it. Page locking is not required, however, if the GPU can use host page tables or other mechanisms to access pageable memory [7,57].

DMA in this context refers to the ability of individual SIMD lanes to access host memory directly without sending requests to the CPU; we do not explicitly use DMA controllers.

We were unable to find reliable published data for the latency of DMA to mapped host memory. Hence, we conducted our own latency measurements as part of this work. Our measurement code can be seen in Listing A8. The resulting values for shared and global memory latencies were consistent with previous findings [40,53], indicating that our measurements are representative.

Likewise, concrete latencies for register and L3 cache accesses are rarely reported in literature. While [53] and [40] extensively analyze the GPU memory hierarchy, neither specify a register or L3 cache access latency. The only source we could identify is a forum post by Sylvain Collange, which suggests that register access latency ranges between 2 cycles and 12 cycles depending on the size and number of operands [2]. However, we did not find any information about L3 cache latencies.

Name	**Size**	**Latency**	**Shared Among**
Registered Host Memory	8-1024GB	1300 - 2500 cycles	System
Global Memory	8-188GB	200 - 1000 cycles	All Threads
Local Memory	<8-188GB	200 - 1000 cycles	Thread
L3 cache	~256MB	unreported	All Compute Units
L2 cache	~4MB	200 cycles	All Compute Units
Shared Memory / L1 Cache	≈10-100KB / CU	20 - 30 cycles	Block / Compute Unit
Registers	≈10-100KB / CU	2-12 cycles	Thread

Table 1 : GPU memory hierarchy showing typical sizes, access latencies, and sharing levels. Most measurements are from [2,6,40,53], and we conducted our own host DMA latency measurements as part of this work.

2.2. HIP (Heterogeneous-Compute Interface for Portability)

HIP [4] and CUDA [54] are C++-based programming interfaces for writing GPU kernels and managing device execution. While CUDA can only be used on NVIDIA GPUs, HIP mainly targets AMD GPUs but also supports NVIDIA.

Programs running on the GPU are called *kernels*, and each SIMD lane behaves similarly to a thread with its own program counter and variables. This execution model is referred to as single instruction multiple thread (SIMT) by AMD [8]. The GPU is often referred to as the *device*, and the CPU side as *host*.

Lanes are organized into three hierarchical levels. At the lowest level, individual lanes are called *threads* on NVIDIA and AMD hardware, while in OpenCL, they are known as *work-items*. Groups of 32 or 64 lanes form a *warp* (NVIDIA) or *wavefront* (AMD), which executes instructions in lockstep. Multiple warps or wavefronts make up a *block* (NVIDIA) or *workgroup* (AMD/OpenCL), where threads can communicate through shared memory. At the top level, all blocks launched for a kernel form a *grid* (AMD/NVIDIA) or *NDRange* (OpenCL), representing the entire collection of parallel executions for that kernel [43,58,63]. Table 2 provides a comparison of terminology across GPU programming models, including both software execution hierarchy and hardware-level execution units.

In the most recent version of *Introduction to the HIP Programming Model* [8], HIP adopts some of CUDA's terminology, namely *Warp*, *Block*, and *Grid*. Accordingly, this thesis also uses the newer HIP terminology for consistency.

All threads belonging to the same block are always executed on the same compute unit [8]. This also means that different blocks can only communicate via global memory since they are not necessarily located on the same compute unit, even if they belong to the same grid. Furthermore, the size of a block and the number of blocks on the same compute unit are limited by the number of warps that can reside concurrently on the same compute unit. This in turn is limited by the number of registers and amount of shared memory each warp uses [5]. The number of warps that can concurrently reside on the same compute unit is called occupancy [5].

Warps on the same compute unit can be efficiently context switched because their complete context is stored inside the compute unit. Blocks, on the other hand, cannot be context switched or preempted. Thus, if no compute unit has sufficient free resources to accommodate a specific block, that block needs to wait for other blocks to finish execution [5].

OpenCL	**CUDA**	**HIP**	**Description**
Compute Unit	Streaming Multiprocessor	*Compute Unit*	A SIMD processor
Work-Item	Thread	*Thread*	A single SIMD lane
Sub-Group (no real name)	Warp	Wavefront / *Warp*	Groups of 32 or 64 lanes, which execute instructions in lockstep.
Workgroup	Thread Block	Workgroup / *Block*	Multiple warps or wavefronts, where threads can communicate through shared memory.
NDRange	Grid	*GPU Kernel* / Grid	All blocks or workgroups launched for a kernel, representing the entire collection of parallel executions for that kernel

Table 2 : Terminology comparison across GPU programming models, including thread hierarchy and hardware execution units. The terminology used in this thesis is cursive.

2.3. Persistent Kernels

Traditionally, GPU kernels are executed with a single purpose; thus, many kernels are executed. This can lead to worse performance due to kernel startup latencies and data transfers that are not overlapped with kernel execution [27,44]. Furthermore, the programmer needs to rely on the GPU hardware scheduler and, therefore, has little control over which blocks are executed if more blocks are launched than the GPU can concurrently execute.

In contrast, *persistent kernels* do not return immediately after completing an operation but instead contain an infinite loop and wait for new instructions or data. Thus reducing kernel startup overhead, giving the programmer precise control over which blocks are executed, and allowing overlapping of data transfers and kernel execution [44].

2.4. Peer-to-Peer Communication

Modern GPUs allow direct communication to other GPUs via the PCIe bus, NVLink [59] by NVIDIA, or xGMI [9] by AMD. Using direct communication can reduce overhead compared to traditional GPU-to-GPU communication, which involves the data being transferred through host memory [10,60]. With peer-to-peer memory access, direct load and store operations on the global memory of other GPUs also become possible [60].

2.5. Challenges Utilizing Multiple GPUs

Efficiently using multiple GPUs or adopting an algorithm to use multiple GPUs poses several challenges. Each GPU has its own global memory, which is, without peer-to-peer DMA, inaccessible to other GPUs. There are alternatives such as Unified Virtual Addressing (UVA), which relies on page faults to migrate pages, and registered host memory; both can lead to overheads if not used carefully. Furthermore, communication between threads located on different GPUs is more expensive than local communication via global memory. Some operations are also not available over the PCIe bus, such as `atomicOr`, or unreliable, as atomic operations in general over PCIe on NVIDIA consumer cards, which we verified during testing [11]. GPUs from different vendors also only have limited interoperability; as such, we could not find a way to enable peer-to-peer DMA between NVIDIA and AMD cards.

To overlap data transfers and kernel execution, it is usually desirable to submit all kernels and data transfers to HIP/CUDA streams and then let the hardware scheduler manage the execution. On multi-GPU systems, this can lead to imbalance because streams are not shared between GPUs, and once a kernel has been submitted to a stream, it can no longer be migrated or canceled [12]. This is exacerbated if the GPUs do not possess the same compute power or the kernels have irregular execution times. Even if a stream is destroyed, all submitted kernels are still executed by the GPU, which we verified using a small test program shown in Listing B9.

3. Related Work

This section compares MGWS to other GPU work stealing solutions, focusing on aspects such as multi-GPU support, general-purpose workload support, and to what extent the CPU is responsible for distributing tasks. Table 3 categorizes relevant previous work under these characteristics. We also present *Structured-atomic Work Stealing System (SWS)* [23], a work stealing system designed for the CPU in distributed memory environments with DMA, which greatly influenced the design of MGWS.

3.1. Load Balancing

The goal of load balancing is to reduce parallel execution time by evenly distributing work across processing elements. Load balancing approaches are commonly divided into two categories: Static strategies, which calculate a schedule beforehand and then assign the work according to that schedule, and dynamic strategies, where the processing elements can exchange work during runtime.

Static scheduling approaches often yield optimal performance but require prior knowledge of task durations. In contrast, dynamic strategies operate without needing to know task execution times in advance, thus enabling new tasks to be dynamically created at runtime.

For our implementation, we chose the dynamic load balancing strategy of work stealing since literature [65] has shown that GPU work stealing performs similarly to static approaches, without the drawback of needing to know the task durations and calculating a schedule prior to execution. Dynamic systems can also be implemented in a decentralized way, which fits well to a self-organized multi-GPU environment.

3.1.1. Work Stealing

Work stealing is a popular dynamic load balancing strategy not only on CPUs but also on GPUs [20,25,31–33,46,51,65,66].

Typically, in work stealing, every processing element has a task queue from which it executes tasks until the queue becomes empty. To obtain new tasks, a processing element E with an empty queue selects a target processing element T, then E removes some work from T's queue and adds it to its own. The processing element that steals the work is called a *thief*, and the target processing element is called a v*victim*. Random victim selection is usually the preferred approach and has been shown to be optimal by Blumofe et al. [22]. However, alternative strategies may become advantageous when additional information about the workload, data, or system is available, such as knowledge of nonuniform bandwidth or latency between processing elements.

Stealing tasks typically causes overhead. Hence, it is desired to minimize the number of steal operations while keeping all processing elements busy. A common strategy to achieve this is for the thief to steal half of the victim's work [28]. Stealing less increases the likelihood that the thief will exhaust its tasks quicker and needs to steal again, whereas stealing more than half may leave the victim with insufficient work, potentially causing them to become idle and steal sooner [28].

Termination is a non-trivial problem: without a mechanism to detect whether there is still work in the system, processing elements will continue attempting to steal work from one another even when no work remains in the system [28]. In MGWS we compare two termination detection algorithms. The first uses a binary tree and is based on work by Cartier et al. [23]. The second uses an atomic counter, keeping track of the number of idle workers. We present both algorithms in Section 4.6.

3.1.2. Structured-Atomic Work Stealing System (SWS)

After introducing load balancing in general and work stealing as our chosen approach, we now turn to a concrete algorithm that can be adopted to our goal of creating a multi-GPU work stealing algorithm that operates independently of the CPU. In particular, we focus on the Structured-Atomic Work Stealing System (SWS) proposed by Cartier et al. [23], which serves as the primary inspiration for our algorithm.

SWS is a CPU-based work stealing algorithm designed for distributed memory systems with remote direct memory access (DMA). It refines the work stealing algorithm originally introduced by Dinan et al. [28]. We first describe SWS in detail before discussing the reasons for its suitability in our multi-GPU setting.

In SWS, every processing element has its own split circular buffer and 64-bit metadata. The circular buffer is divided into two portions: a *shared* portion, from which other processing elements can steal work, and a *local* portion, which can only be accessed by the buffer's owner. To avoid confusion with GPU shared memory, a memory area shared by one GPU thread block, this thesis refers to the "local" portion as *local* and to the "shared" portion as *public*. A processing element interacts with the metadata and public region of its circular buffer only when its local region is either empty while the public region still holds tasks or vice versa. In the first case, when the local region is empty but the public region still has tasks, work is transferred from the public region to the local one. This operation is referred to as *acquire* by the authors. In the second case, tasks in the local region are made available to others and moved to the public region, which is called *release*. Other processing elements can only steal from the public portion and do not have access to the local portion.

SWS compacts the queue metadata so that a single atomic operation is enough for both checking whether a victim has work and claiming half of that work.

The focus of SWS on distributed memory systems with remote direct memory access (DMA), together with its separation of tasks into a local and a public portion, aligns well with the memory hierarchy of modern GPUs and is a key reason why we base our implementation on SWS. Connecting multiple GPUs via peer-to-peer DMA or host memory closely resembles a distributed memory environment with DMA. Moreover, even a single GPU fits this model because each worker has access to fast local shared memory, while all workers share a global memory space. Having separate public and local portions allows us to take advantage of the hierarchical memory architecture of GPUs by placing each portion at a different level of the memory hierarchy. We discuss our design in more detail in Section 4.2.

3.2. GPU Load Balancing

This section first discusses previous task-specific GPU work stealing solutions. These approaches are usually not easily generalizable to handle general-purpose applications; they nevertheless solve interesting problems such as termination detection or task dependencies, which also general-purpose work stealing algorithms such as MGWS face. This is followed by relevant general-purpose GPU work stealing approaches limited to a single GPU. These approaches are often more independent of the CPU and more decentralized than their multi-GPU counterparts, which aligns with our goal for a decentralized, CPU-independent algorithm. Finally, we visit general-purpose multi-GPU work stealing algorithms, which, in contrast to our approach, are all relying on the CPU for task distribution.

3.2.1. Task-Specific GPU Work Stealing

Load balancing is an important issue in general-purpose GPU programming. A large portion of research focuses on balancing the compute load for specific applications. While these approaches sometimes also use work stealing, they are intended for specific problem domains and are usually not easily generalizable to others, unlike MGWS, which can handle unknown custom user-defined tasks.

GUM [51], for instance, uses a multi-GPU load balancing system designed for graph processing. It leverages work stealing to evenly partition graphs across GPUs. The authors also discuss that, depending on the topology, multiple data paths with different performance characteristics may exist between devices. GUM is based around an iterative execution model. At the beginning of every iteration, a coordinator develops a stealing policy that aims to minimize the end-to-end execution time of the iteration. To that end, the coordinator uses a cost coefficient matrix, which describes the cost for a worker to process an edge located on another worker; this matrix is dependent on the topology. The coordinator may even exclude GPUs from the work stealing process if it deems that communication overhead outweighs the computation time. While interesting for future work, this is out of scope for this thesis and also not required since we evaluate at most two GPUs and use PCIe as the only link between them.

Similarly, Gmys et al. [33] take advantage of multiple-GPU work stealing to solve branch-and-bound optimization problems. Their approach uses a hierarchical work stealing scheme, which prioritizes load balancing inside GPUs over intra-GPU load balancing. Each GPU has a CPU thread, referred to as *GPU-thread*, which is responsible for invoking kernels for the various phases of the branch-and-bound algorithm and stealing work from other GPUs or CPU workers. Thus, the algorithm depends on the CPU for intra-GPU load balancing, which is not the case in MGWS, where each GPU worker can directly steal from workers located on other GPUs, thereby avoiding overheads caused by communication and synchronization with the CPU. Termination detection is done via an atomic counter, which is incremented when a worker becomes idle; the GPU-threads increment this counter on behalf of their GPU. Using a counter for termination detection is similar to one of our termination detection strategies, which also employs an atomic counter to track the number of idle workers. However, since MGWS has no CPU threads managing the GPUs, each GPU worker increments this counter itself when it becomes idle.

Hermann et al. [38] make use of task graph partitioning combined with work stealing to calculate physics simulations on multiple GPUs and CPUs. Their framework reimplements parts of the CUDA API to intercept kernel launches and, if necessary, migrate them to different GPUs, which is out of scope for this thesis but allows for a more natural interface for users. Hermann et al. also implement their own shared memory scheme between GPUs and CPUs, which keeps track of memory accesses and migrates data on demand between devices. This is especially interesting because NVIDIA only released a similar system called *unified memory* several years later in CUDA 6. However, concrete inter-GPU data migration schemes apart from the aforementioned unified memory are out of scope for this thesis. Similar to MGWS, their approach also supports executing tasks on GPUs and CPUs. However, they separate the implementation instead of relying on wrappers around device-only functions, which allows maintaining a single code base for both GPU and CPU if desired. MGWS adopts such a wrapper-based design, reusing many of the abstractions originally introduced in GPU4FS [50].

The approach of Hermann et al. also supports task graphs to statically declare dependencies between tasks, whereas in MGWS we model task dependencies by dynamically spawning subtasks at runtime after a task has finished its computation. Maintaining a task graph is more difficult in a general-purpose setting compared to a specific application such as physics simulation, where more is known about specific tasks. Modeling static task dependencies is nevertheless an interesting topic for future work.

3.2.2. Single-GPU Load Balancing

Having discussed task-specific multi-GPU work stealing schemes, we now focus on general-purpose single-GPU load balancing to explore approaches applicable to our use case: existing literature on general-purpose work stealing is often limited to single-GPU environments, but the CPU involvement varies. In this section we first look at a single-GPU work stealing approach by Gad [31], where the CPU is responsible for distributing tasks among GPU workers. This is followed by single-GPU work stealing approaches, which are more independent of the CPU, following a decentralized design.

MGWS is not limited to a single GPU but can utilize multiple GPUs by facilitating communication via peer-to-peer DMA or host memory. Our approach also reduces GPU-CPU communication and synchronization overheads by letting GPU workers distribute work among themselves independently of the CPU.

Many GPU load balancing systems are implemented using CUDA limiting them to NVIDIA GPUs. By using ROCm HIP and wrappers around device-only functions, our approach cannot only run on NVIDIA GPUs but also on CPUs and AMD GPUs, which we discuss further in Section 5.8.

Gad [31] proposes a single-GPU work stealing framework for iterative tasks, where workers can run on either the CPU or GPU. Task queues are maintained in host memory, allowing GPU and CPU workers to steal work from one another. We adopt a similar strategy by storing public task queues in host memory when peer-to-peer DMA is not used to facilitate communication between GPUs; otherwise, we store public task queues in global GPU memory.

At the beginning of every iteration, a CPU thread referred to as *main thread* chunks the tasks and distributes the chunks to the worker queues. For GPU workers, every thread is mapped to one task in the chunk. Then the main thread launches the GPU and CPU workers. Once all workers have completed their tasks, the main thread collects any new tasks that were created during the last iteration and commences the next iteration. In MGWS every task is also capable of spawning new subtasks, but unlike Gad's framework, our approach is not confined to an iterative execution model and instead has an initial task set that is loaded by the GPU workers independently of the CPU. Afterwards, if workers are idle, a GPU worker can steal tasks from other GPU workers, removing any involvement of the CPU in distributing tasks.

Tzeng et al. [66] present a single-GPU load balancing system using work stealing independent of the CPU. They are also explicitly mentioning persistent kernels as the execution strategy for their GPU workers, which we adopted in MGWS. Similar to our approach, MGWS, Tzeng et al. divide the GPU up into workers using persistent kernels. Their system is designed for pipelines where each worker may receive a different or unpredictable amount of work due to irregular workloads. While MGWS is not specifically designed with pipelines in mind, the creation of subtasks allows for pipeline-structured execution. Unlike our approach, they manage work at the warp level. By balancing work on the block level, we aim to give the user more options. Users may choose to take advantage of shared memory among warps within the same block or define the block size to be equal to the warp size, effectively enabling warp-level work balance if desired.

Cederman et al. [24] implement a lock-free work stealing algorithm based on work by Arora et al. [21], where every worker owns a double-ended queue. They store the double-ended queue in global memory so it can be accessed by other workers. We use two separate queues instead of a single queue, which allows us to take advantage of the GPU memory hierarchy by storing tasks that will be executed next in faster shared memory and tasks that can be stolen by others in global memory.

In their approach, Cederman et al. try to minimize the use of atomic operations due to performance concerns, which we do not consider an issue given [49]. The authors represented each GPU worker with a block, as we do in MGWS, which allows fast communication and synchronization among threads belonging to the same worker.

In their work, Chatterjee et al. [25] present a *finish-async* style API for GPU kernels, which allows the launching of asynchronous subtasks and waiting for their completion. They also represent workers as blocks that have their task queues reside in global memory, thereby increasing latency compared to storing tasks in shared memory. With MGWS, it is possible to launch new asynchronous subtasks, but we provide no way of waiting for individual tasks because this was not in the scope of our approach. A possible way to implement this feature using a global registry is discussed in Chapter 7. However, Chatterjee et al. mention that they also only thoroughly tested waiting for all async tasks rather than individual tasks. They also claim that the only way for blocks to synchronize is through multiple kernel launches, which is not true anymore because blocks can communicate and synchronize via global memory or even registered host memory.

While their approach supports multiple devices, work stealing is limited to workers located on the same device. Unlike MGWS, inter-GPU load balancing is not possible using their approach, and tasks are instead evenly distributed among devices before execution, which may lead to imbalance if task execution times vary.

Chatterjee et al. use a global task counter for termination detection. This may lead to the counter being more contested than our atomic counter-based termination detection, which counts idle workers rather than remaining tasks.

Lastras-Montano et al. [45] propose a work stealing system using multiple task queues on different levels of the GPU memory hierarchy, thus taking full advantage of the low-latency shared memory. Their approach represents workers as warps and allows workers to either steal from other workers located on the same compute unit via shared memory or from workers located on remote compute units via global memory. We represent workers as blocks, which gives the user more flexibility over the number of threads per worker. Thus, however, stealing in shared memory is not possible in MGWS; we instead use queues in shared memory to store tasks that cannot be stolen and are executed next. Allocating queues for each warp in shared memory may also consume more resources than only having one queue per block. Unlike most other work stealing approaches, the authors also investigate whether consecutive victim selection, where a thief iterates through all queues until it finds a queue containing tasks, is better than random victim selection, used by most work stealing systems. They conclude that random victim selection yields better results because it distributes work more evenly across workers.

Lastras-Montano et al. require locks to avoid races on the task queues compared to our lockless strategy. Additionally, their termination detection mechanism lets workers exit once they fail a certain number of steal attempts in a row. Using such an approach avoids termination detection-related communication overhead between workers, as termination decisions are made independently by each worker. However, it can lead to severe imbalance because phases with less work may cause a high number of workers to exit, which then in phases with more work are unavailable.

In [65], Toss presents three GPU load balancing strategies: static load balancing, list scheduling, and work stealing. They conclude that work stealing is the most generic scheduling strategy, performing well on both regular and irregular workloads. We focus on the work stealing approach because this is also the policy used in MGWS. Work stealing is more general than static load balancing because it does not require prior knowledge of task execution times. Moreover, its decentralized nature is better suited to avoiding bottlenecks than a centralized approach such as list scheduling. Toss implements workers as blocks and lets every worker maintain a double-ended task queue in global memory, unlike our approach, which utilizes multiple queues per worker residing in both global memory and low-latency shared memory. In contrast to our lockless queue approach, Toss uses locking to prevent data races when stealing tasks. Similar to Chatterjee et al. [25] Toss relies on a task counter for termination detection, which may lead to higher contention than our atomic-counter-based approach, which counts idle workers instead of tasks.

3.2.3. Multi-GPU Work Stealing

Despite multi-GPU workloads receiving considerable attention, task assignments are always handled by the CPU [20,32,42,46]. We were unable to find any prior research that utilizes multi-GPU work stealing while operating independently of the CPU. That said, we were frequently not able to determine the exact extent of CPU involvement in both single- and multi-GPU approaches.

None of the presented approaches utilize peer-to-peer DMA for direct GPU-to-GPU communication; we intend to exploit peer-to-peer DMA to enable GPU workers to steal from other GPU workers directly without having to store queues in host memory. Similar to the single-GPU solutions, most approaches use CUDA and are thus limited to NVIDIA GPUs, although some also support CPU workers. In contrast, MGWS can support both NVIDIA and AMD GPUs by using ROCm HIP instead of CUDA. CPU workers are also supported and share a code base with GPU workers by utilizing wrappers around device-only functions, which we discuss more in Section 5.8.

Lima et al. [46] implement multi-GPU work stealing orchestrated by the CPU. Their approach runs a CPU thread for each GPU that takes part in the work stealing algorithm on behalf of their GPU. MGWS does not require a dedicated CPU thread for each GPU, as GPU workers steal work autonomously. Each task is represented by one or more kernels. Thus, in contrast to our approach, no persistent kernels are used, and no further load balancing happens on the GPU.

Lima et al. did not take advantage of peer-to-peer DMA because, at the time, this feature was unreliable, according to the authors. MGWS is designed to leverage peer-to-peer DMA to enable direct data transfer and steal operations between workers running on different GPUs. However, we were unable to evaluate this communication mechanism, as it relies on coherent atomic operations over peer-to-peer DMA, which were not available in our evaluation setup.

In their paper, John et al. [42] propose a system for load balancing using work sharing on multiple GPUs in the context of dataflow programming. The authors rely on PaRSEC, a runtime for dataflow applications, to model dependencies between tasks. MGWS does not explicitly model dependencies between tasks but allows tasks to spawn new tasks, which can be used to model simple dependencies. A more sophisticated way of modeling inter-task dependencies is out of scope for this thesis.

Albeit more recent than the approach by Lima et al. [46], their system still relies on GPU management threads running on the CPU to orchestrate work stealing. The manager thread keeps track of the load of its GPU and steals work from other GPU task queues in case of starvation. When selecting tasks, they also consider data locality and whether tasks access the same data. Our approach is more decentralized, where workers running on the GPU orchestrate the work stealing process themselves instead of relying on one CPU management thread per GPU.

Similar to our approach, they support CPU workers and GPU workers running on AMD or NVIDIA GPUs. However, they don't provide much information on how they facilitate this.

Data management is handled by providing each GPU with its own copy of the data, along with an associated version number. They also ensure that only one task at a time can modify the data and that any modification results in the version number being incremented. The runtime automatically loads the latest version when a subsequent task requires accessing that data. While explicit task data management is out of scope for this thesis, their version-based approach is intriguing as long as multiple tasks do not need to write to the same data.

Arafat et al. [20] implement Scioto-ACC, a work stealing algorithm spanning multiple GPU-CPU nodes. Each node has a scheduler that is responsible for transferring data between CPU memory, GPU memory, and a global address space. It also steals tasks from other nodes and starts execution of tasks on GPU or CPU workers. The authors use pipelining techniques to overlap execution and data transfers. We do not implement a centralized scheduler responsible for distributing tasks; instead, we utilize a decentralized approach where each worker may steal from any other worker. A decentralized strategy was chosen to avoid a work arbiter becoming a bottleneck and to reduce communication and synchronization overheads. Exploring how MGWS scales to multiple nodes is beyond the scope of this thesis; we focus our attention on single-node systems with multiple GPUs. Nevertheless, MGWS could be incorporated into a hierarchical load balancing approach, where it handles intra-node load balancing across multiple GPUs, while a separate scheme manages inter-node load balancing.

Scioto-ACC can handle task data management and lets the user define functions to transfer data from and to GPU memory. Data for GPU tasks is always copied twice, once from global address space to host memory and then again to GPU memory. Thus, sharing data between tasks is not possible because data is always copied. Not sharing data between tasks makes sense on multi-node systems where a GPU might not be able to access memory on other nodes, but in a single-node setting, sharing data between tasks may be desired by the user.

Arafat et al. let the user define whether a task is better suited for GPU or CPU execution and take this into account when scheduling the tasks. MGWS does not take this into consideration because without a centralized scheduler, it is more challenging to ensure that tasks stay on the CPU or GPU. This is, however, an interesting subject for future work.

In [32], Gautier et al. present a work stealing scheduler for multi-CPU and multi-GPU systems, which also takes data locality into account. The authors support GPU and CPU execution by requiring separate task bodies for both. The GPU task body is also responsible for launching the GPU kernel. Instead of launching a kernel per task, MGWS leverages persistent kernels, which load and execute tasks independently of the CPU, thereby avoiding potential overheads of synchronization or communication between the GPU and CPU. The authors allocate a queue for each CPU or GPU thread to which other threads can push tasks. However, unlike our approach, a GPU worker does not manage its own queue; instead, the runtime is responsible for transferring necessary data to and from the device and starting execution of the GPU kernel.

Gautier et al. manage task data on the GPU using a software cache with a least recently used replacement policy. The runtime transfers data from host memory to global memory if a GPU task requires it, and if the GPU has no space left, it evicts old memory blocks. Maintaining a software cache is an interesting solution for GPU data management but out of scope for this thesis.

When deciding where to execute a task, the scheduler uses two heuristics: the first prioritizes a target where a high number of input bytes are already present, and the second tries to reduce invalidations by preferring targets with valid *write* or *exclusive* copies of data. MGWS currently does not explicitly manage task data, and we also do not have a centralized scheduler that could map a task to a specific worker based on the data locality. We instead opt for a more decentralized approach to reduce the risk of a single scheduler becoming a bottleneck.

Publication	Multi-GPU	GPU-Orchestrated	General-Purpose
Lima et al. [46]	✅	❌CPU thread per GPU	✅
Scioto-ACC [20]	✅	❌CPU scheduler	✅
John et al. [42]	✅	❌CPU thread per GPU	✅
Gautier et al. [32]	✅	❌Task as kernel	✅
GUM [51]	✅	✅	❌Graph processing
Hermann et al. [38]	✅	✅	❌Physics simulation
Gmys et al. [33]	✅	❌CPU thread per GPU	❌Branch-and-bound
Tzeng et al. [66]	❌	✅	✅
Cederman et al. [24]	❌	✅	✅
Chatterjee et al. [25]	❌No inter-GPU stealing	✅	✅
Toss [65]	❌	✅	✅
Gad et al. [31]	❌	❌CPU task distribution	✅
MGWS	✅	✅	✅

Table 3 : An overview of previous work and our approach, comparing three key capabilities: support for multiple GPUs, GPU-side work distribution independent of the CPU, and the ability to handle arbitrary tasks.

4. Design

This section presents the design of MGWS, our work stealing system. We describe each component and discuss the design choices we considered during the planning phase of MGWS. Each subsection addresses a different aspect of MGWS.

We represent every GPU worker as a block within a persistent kernel (Section 4.1), running its own instance of the work stealing algorithm (Section 4.2) independently and without CPU involvement. MGWS supports lockless stealing, which involves performing atomic fetch-and-add operations on a target queue's metadata. This can lead to overflows, which we must detect or prevent (Section 4.3). Our work stealing algorithm is based on SWS [23], but we made a number of alterations to better fit a multi-GPU environment (Section 4.4). Tasks, encapsulated in a task struct, can be stolen by workers (Section 4.5). MGWS detects global completion by running a decentralized collective termination detection algorithm (Section 4.6). Workers may be located on remote GPUs and communicate via peer-to-peer DMA or registered host memory (Section 4.7). Some tasks may depend on additional data that cannot be stored directly inside the task struct. In such cases, workers may need to map or load necessary data into global memory before execution (Section 4.8).

4.1. Persistent Kernels

Compared to previous GPU work stealing implementations, our GPU workers steal work without CPU involvement. However, only the CPU can launch new kernels on the GPU. To reduce this reliance on the CPU, we refrain from mapping each task to its own kernel, as seen in [46], [32], and [20], but instead implement GPU workers using persistent kernels. Each block part of the persistent kernel executes an instance of the work stealing algorithm and only exits when no tasks remain. Using persistent kernels provides a few advantages: not launching a new kernel per task reduces startup overhead, alleviates CPU involvement when executing new tasks, and gives more control over which blocks are concurrently executed on the GPU instead of relying on the hardware scheduler [44]. MGWS aims to utilize the whole GPU; as such, MGWS launches as many blocks as can be concurrently executed, where each block is a worker with its own task queue. In Section 6.4 we evaluate whether utilizing the whole GPU with persistent kernels running our work stealing algorithm leads to thermal throttling.

4.2. Work Stealing

We base MGWS on the SWS algorithm proposed by Cartier et al. [23]. SWS is designed for distributed memory systems with remote direct memory access (DMA), supports lockless stealing via a single atomic fetch-and-add, and separates a worker's task buffer into two queues. These properties align well with our goal of developing a CPU-independent multi-GPU work stealing algorithm. This section outlines how they influence the design of MGWS.

Distributed memory systems with remote DMA: A multi-GPU environment with peer-to-peer DMA or host memory to facilitate communication closely resembles a distributed memory system with DMA. Moreover, even a single GPU fits this description since workers have access to fast local shared memory and shared global memory.

Two task queues: The work stealing algorithm allocates two queues per worker: a private local queue containing the tasks that will be executed next and a public queue containing tasks that can be stolen by other workers. We aim to leverage the GPU memory hierarchy by placing each queue at an appropriate level. Global memory is significantly slower than shared memory, with access latencies ranging from 200 to 1000 cycles compared to just 20 to 30 cycles for shared memory [53]. Storing tasks in faster shared memory is therefore beneficial as it is fast and accessible by all warps of the same block. However, storing both queues in shared memory is not feasible since shared memory is rather small, in the range of tens to hundreds of kilobytes, and cannot be accessed by other blocks. We considered two approaches on how to adapt SWS to take advantage of the fast shared memory while still allowing other workers to access the public queue:

1. Two separate circular buffers, one residing in global memory containing the public queue and the other in shared memory containing the local queue.
2. A single circular buffer containing both queues and shared memory as a cache for parts of the local queue.

We decided to implement the first approach because it is well suited for host memory-based and peer-to-peer DMA-based communication. Next we discuss both approaches in more detail.

The first approach uses two separate circular buffers as originally proposed by Dinan et al. [28], where the public buffer is located in global memory and the local buffer is located in shared memory, as shown in Figure 1. Storing the local portion in shared memory is possible because the local buffer is only accessed by the owner of the buffer. Compared to using a single circular buffer, this approach is simpler to implement and allows public queues to be shared among multiple workers. While increasing the number of workers sharing a public buffer reduces the global memory footprint, it also raises contention since acquire and release operations require locking. The downside of using two queues is that releasing and acquiring tasks involves copying them between queues.

The second approach stays closer to SWS by using a single circular buffer, located in global memory per GPU worker, and replicating parts of the local portion into shared memory, depicted in Figure 2. Using a single buffer has the advantage that the entire local task portion does not need to reside in shared memory, and no tasks are copied on release. However, this approach is more complex and results in higher overall memory usage due to data replication and the inability to share public queues.

Lockless stealing: A thief determines whether a specific public queue contains any tasks, and if so, claims a portion of them by performing a single fetch-and-add on the 64-bit queue metadata, enabling lockless stealing. Using atomics is also possible in our environment because GPUs can perform atomic operations on all levels of the memory hierarchy, i.e., shared memory, global memory, global memory from other GPUs via peer-to-peer DMA, and even registered host memory. Similar to SWS, MGWS is also structured in epochs to avoid workers having to wait for steals to complete before manipulating their public queues. An epoch describes the current state of the public queue. It contains the number of steal attempts, the initial number of tasks at the start of the epoch, and the location of the first task in the queue, referred to as the *epoch tail*. A new epoch starts whenever the owner of the public queue transfers tasks from or to its public queue. Steal attempts made in one epoch can overlap with other epochs, but the number of concurrent epochs is limited. Thus, when a worker enters an epoch again, it needs to wait until all outstanding steals in that epoch are completed. The current setup uses two epochs, as does SWS [23], which suggests that two are sufficient to minimize waiting times.

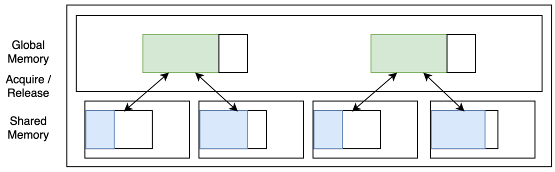

Figure 1 : Two separate buffers are shown. The local buffer in blue is located in shared memory, and the public buffer in green resides in global memory. The local tasks are not stored in global memory; thus, data needs to be moved between buffers on acquire and release. Every public buffer is shared between two workers.

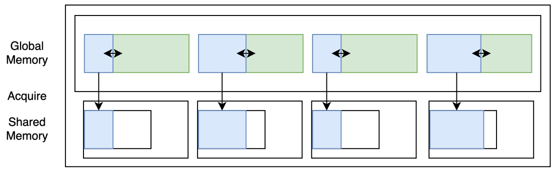

Figure 2 : The split circular buffer located in global memory. The public portion is highlighted in green, and the local portion is highlighted in blue. The local data is replicated into shared memory and only copied from the global memory on acquire. Release does not necessitate any data movement and only shifts the border between local and public partitions.

4.3. Steal Counter Overflow Prevention

Every steal attempt increments the *attempted steals* field of the metadata. Incrementing the counter can lead to an overflow, which in turn can lead to the same tasks getting stolen multiple times. SWS prevents overflows by letting every worker maintain a bitmap, where each bit represents a worker and whether that worker still had work when last targeted by a steal. Whenever a worker tries to steal from a target where this bit is set, the worker first performs an atomic fetch to determine if the target has any work. This is followed by an atomic fetch-and-add on to the attempted steal counter if the target has work that can be stolen. On a successful steal, the bit inside the bitmap for that target is also reset. Checking the bitmap before incrementing prevents overflows of the attempted steal counter because before incrementing the counter a second time, a worker checks whether work is available.

We propose our own approach using overflow flags with the aim to reduce the number of required atomic operations. However, our approach still has some issues, which we discuss later in this section and which we were not able to alleviate. These issues led us to adopt the strategy used by SWS instead.

Our proposed implementation tracks overflows by adding an overflow bit for each of the two epochs to the metadata. To make room for these bits, the attempted steals field is reduced by two bits, from 24 to 22 bits. The owner of the queue resets the corresponding overflow bit on entering the epoch. Each worker attempting to steal tasks increments the attempted steals field and checks the overflow bit. A steal attempt only succeeds if the overflow bit of the current epoch is unset and the public queue still contains tasks. When a worker determines that no tasks remain, it sets the overflow bit for the epoch in which the steal is attempted.

We avoid overflows before the overflow bit is set by limiting the number of workers N: The overflow happens after 2^{22} increments; however, in the worst case the attempted steal counter is 18, one less than the initial task field width, before the first worker tries to set the overflow bit. Thus, when limiting the number of workers to $2^{22} - 18 = 4\,194\,285$. Even if all workers try to steal concurrently from the same queue and thereby increment the attempted steal counter, the counter does not overflow, since at least one worker must set the overflow bit before any worker can attempt another steal.

Another point of consideration is to use two overflow bits, one for each epoch. The motivation of this design is to avoid a race that occurs when the epoch changes once in the interval between the worker observing that no tasks remain and it setting the overflow bit. If a single overflow bit were shared across both epochs, a worker could incorrectly mark an overflow in an epoch where no overflow occurred. However, even with two overflow bits, a worker may still incorrectly set the overflow bit if the epoch changes twice in that interval, since the system wraps back to the original epoch after two epoch transitions.

We considered multiple ways to solve this issue, but none were sufficient to eliminate all potential race conditions while keeping lock-free stealing:

- Using atomic compare-and-swap instead of atomic OR does not solve the issue because a worker still cannot determine whether the epoch changed twice or if it is still the same epoch.
- Introducing a lock bit for stealing tasks solves the issue since the owner of the queue needs to wait until all steals are completed; this, however, requires locking by thieves, which we want to avoid.
- Waiting for all steal attempts to complete before reentering the epoch would allow workers to set the overflow bit before finishing the steal attempt. However, this is not feasible in the current implementation, where we can only wait for workers that successfully stole tasks. Workers that fail to steal because the public queue became empty are not waited for, yet they still set the overflow bit. In the following, we discuss (1) how the algorithm could be modified to wait for all steal attempts, regardless of their success, and (2) the implications of restricting overflow bit updates to only successful steal attempts and waiting only for those workers.
    1. One way to wait for workers whose steal attempts fail is to introduce a separate counter that is incremented before a worker performs an atomic fetch-and-add on the attempted steal counter and decremented when a worker completes the steal attempt. Thus allowing the owner of the public queue to wait until all steal attempts are completed before reentering the epoch. However, instead of performing two atomic operations on the new counter, a worker could simply perform an extra atomic fetch on the existing attempted steal counter and prevent overflows with fewer atomic operations. Furthermore, this approach incurs more atomic operations than the strategy used in SWS.
    2. Restricting overflow bit updates to only successful steal attempts implies that the last worker to successfully steal tasks must set the overflow bit; otherwise, some tasks may never be stolen because subsequent workers will incorrectly assume that the counter has overflowed. This requirement necessitates the last successful thief to set the overflow bit before marking its steal as complete. While this solves the original race, it introduces a new one since the attempted steal counter may overflow in the interval between the last worker detecting that it should set the overflow bit and it actually performing the update.

Beyond correctness, our proposed solution relies on atomic OR operations to efficiently set the overflow bits; while this works well when only using one GPU, atomic OR is not supported over PCIe [11]. Consequently, we must fall back to atomic compare-and-swap, which is supported over PCIe but can cause a high number of retries, particularly when the counter is contested. The lack of atomic OR over PCIe, together with the unsolved correctness issues, prompts us to instead adopt steal damping as proposed by Cartier et al. [23].

Category	Structured-atomic Work Stealing System	MGWS
Target system	Distributed memory with DMA	Multi-GPU considering GPU memory hierarchy and peer-to-peer DMA
Task queues	Single split circular buffer with public and local partition	Two circular buffers: Public queue in global memory Local queue in shared memory
Metadata	Compact metadata single 64-bit atomic	Similar metadata optionally with overflow bits
Steal completion	Tracks steal completion through bitmap	Two atomic counters: Steals in progress Steals completed
Queue sharing	Not applicable due to singular circular buffer	Optional sharing of public queues

Table 4 : A high-level comparison of the main differences between our work stealing approach and SWS.

4.4. Differences to SWS

MGWS is inspired by SWS [23], but we modify several aspects to better suit our goal of enabling work stealing in a multi-GPU environment. Table 4 provides a high-level overview of the key differences between our approach and SWS. The remainder of this chapter discusses the differences in more detail.

Queue circular buffer: Each worker maintains a local and a public queue that contain all tasks owned by that worker. Instead of using a single circular buffer for both the local and public queues, as SWS does, MGWS uses two separate circular buffers, one for each queue. Using two separate buffers follows the approach proposed by Dinan et al. [28], on which SWS is based. The public queue is located in global memory or host memory, depending on the intra-GPU communication scheme, while the local queue is allocated in shared memory. Storing the local queue in shared memory allows tasks that are executed next and cannot be stolen by other workers to be accessed more quickly, while the remaining tasks are kept in slower global memory accessible to other workers. Separating the queues, however, complicates the release and acquire operations, since workers must move tasks between the two buffers. In SWS, these operations only involve updating the split point between the queues. In case of a release, no locking is required in SWS because the public queue is always empty before the operation, so no other worker can initiate a steal. The acquire operation, however, must lock the public queue. Unlike SWS, MGWS must support releasing tasks into a non-empty public queue when the local queue cannot hold any more tasks. Hence, a worker also needs to lock the public queue during a release operation. Locking during release is unnecessary in SWS because the local task queue portion can grow to meet demand as long as the circular buffer containing both queues still has enough free space.

Metadata: The metadata is part of the public queue. It stores information about the number of tasks and their position in the buffer. Workers use the metadata to determine which tasks they can steal or acquire. Our metadata is similar to that of SWS. We only conducted two changes:

adding two overflow bits in case the overflow bit-based overflow detection strategy is used and dividing the epoch counter into a lock and an epoch bit. The overflow bits are only present if our overflow detection strategy is used instead of the steal damping mechanism proposed by SWS. To add the two overflow bits, the size of the attempted-steal counter is reduced by two bits.

Dividing the epoch counter into a lock and an epoch bit is only a structural change, since SWS already uses two bits to encode the two epochs. It defines the public queue as invalid when the epoch counter is greater than the number of epochs. The invalid state effectively acts as a lock because it prevents other workers from stealing tasks so the owner can freely modify the queue contents and metadata. Thus making it equivalent to using one bit for the epoch and one bit as a lock indicator. We aim to make this explicit by introducing a dedicated lock bit.

Steal completion tracking: A public queue owner is only allowed to overwrite stolen tasks in the queue once they have been fully transferred. Workers do not lock the public queue on steals; hence, a mechanism is needed to detect whether a given task, or region of tasks, is still in transfer or can be safely overwritten. SWS tracks completed steals by assigning each task region one of four states: available (A), claimed (C), finished (F), or invalid (I). When a new epoch begins, the worker iterates over every region of all epochs to determine the longest sequence of finished tasks starting at the tail. These consecutive task regions can then be overwritten by new tasks. If all epochs contain unfinished steal operations, the worker needs to spin until at least one epoch is fully completed.

MGWS tracks steal completion using two counters per epoch: one counts the number of tasks marked for stealing, and the other counts the number of tasks whose steal process has completed. Using two counters offers some advantages: it is simpler, uses less storage because no per-region status bits are needed, and avoids scanning through completion arrays to determine newly available space.

The approach of SWS, on the other hand, has the advantage that partial epochs can be freed, which is useful because SWS stores both local and public queues in the same buffer. Freeing space in the public queue portion sooner enables the worker to enqueue more tasks into the local queue. Our algorithm uses separate buffers for each queue. Thus, as long as the local queue still has space for new tasks, freeing space in the public queue offers no benefit. Additionally, SWS only begins a new epoch during an acquire operation. In contrast, our design must also start a new epoch during release operations, allowing us to free space before transferring tasks from the local to the public queue. Furthermore, task transfers are fast if the workers are located on the same GPU, and they remain fast across GPUs when peer-to-peer DMA is utilized. This helps to complete epochs quickly, which in turn increases the likelihood that epochs are fully completed when an epoch transition occurs.

Sharing of public queues: Our algorithm allows public queue sharing among workers, which is only possible because MGWS uses separate circular buffers for each queue. SWS does not allow public queues to be shared. Sharing public queues increases contention because multiple workers use acquire and release on the same queue but reduces global memory usage. To support shared queues, it must be possible to release tasks into non-empty public queues, which we already implemented as part of enabling separate circular buffers. Additionally, we treat the lock bit in the metadata as a real lock, meaning the owner of the public queue must check whether it is locked before acquiring it. A worker may need to spin until the other owner releases the lock.

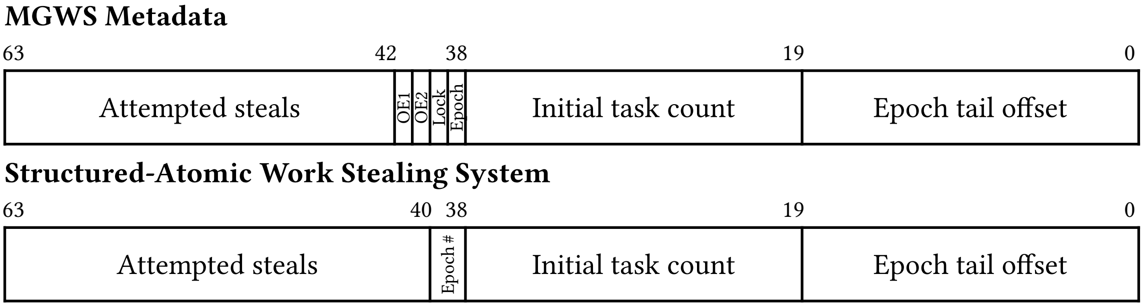

Figure 3 : Shows the public queue metadata of our algorithm and SWS [23]. The OE1 and OE2 flags are the epoch overflow bits and indicate a potential overflow of the attempted steals counter for an epoch. They are only present if the overflow bit-based overflow detection strategy is used instead of the steal damping mechanism, which is also used by SWS.

4.5. Task Struct

The task struct represents a task in memory. It therefore needs to contain everything necessary to run the task:

- A reference to the function to be executed, which we call *lambda*
- The function *parameters*, local to this task
- Information about required data, which may be accessible by all tasks, e.g., buffers

To enable efficient stealing of tasks, the struct should be concise and have a fixed size.

The function parameters are stored directly in the task struct. However, some tasks need to access large amounts of data, which might be shared with other tasks and therefore cannot be stored inside the task struct. Moreover, depending on the location of the data and the data management strategy, GPU workers must copy data to the device before executing a task. Thus, MGWS not only needs to know where the data is located but also the size and access rights, i.e., if the data is only read or also written to. This information is wrapped inside a `DataDescriptor` and stored inside the task struct. However, it is only possible to store a fixed number of `DataDescriptors` inside the fixed-size task struct. Any `DataDescriptor` that does not fit into the task struct is stored in host memory or global memory and loaded by the worker on task execution.

We considered three options for passing the lambda between CPU and GPU and chose the third as our preferred method:

1. Storing the lambda inside `std::function` would be the most convenient option; however, `std::function` is not available on the GPU, and porting it would be required.
2. Store a raw C++ lambda inside the task struct using the type of the lambda as a template parameter. After copying the raw task struct data from CPU to GPU, the vtables created by the CPU cannot be reused on the GPU for polymorphism. It is therefore necessary to store a type tag in the task struct which is mapped to the type of the corresponding C++ lambda. As a result, the lambda needs to be known at compile time.
3. Encapsulate the lambda as a member function of a task functor struct. Using a functor is similar to the raw C++ lambdas because a mapping from a given tag to the functor type needs to be created, but it additionally allows the explicit storing of function parameters inside the functor struct. The functor is stored directly in the task struct. Furthermore, the space for function parameters is limited to a predefined number of bytes to ensure fixed-size task structs. Any additional parameters must be stored outside the task struct and referenced by a

`DataDescriptor`. A user of MGWS can define a new task by deriving it from the base struct `TaskFunctor` and implementing it.

We chose to store the lambda as a member function in the task functor because it enables explicit storage of the function parameters in contrast to a raw C++ lambda. It also provides a clear interface to implement new custom tasks. We decided against porting `std::function` as it was deemed out of scope for this thesis.

4.6. Termination Detection

Efficiently detecting whether there is work in the system is a nontrivial problem [28]. We compare two approaches, one based on a binary tree and the other based on incrementing an atomic counter. In the binary tree, every GPU worker can vote on whether it still has work. The root then decides whether any work is remaining and shares its result with all other workers [28]. In the second approach, every GPU worker who is idle increments a counter in global memory and decrements it again if it regains work. Thus, if there is no work left, the counter will be equal to the number of GPU workers.

The second approach is simpler than the first but also has asymptotically a worse runtime $O(n)$ compared to $O(\log n)$ for the binary tree, where n is the number of GPU workers. However, GPU atomics are highly optimized; thus, the actual runtime is comparable to or faster than the binary tree [49]. In Section 6.6, we show that the atomic counter causes less overhead than the tree-based approach when used in global memory.

4.7. Multi-GPU

To eliminate CPU intervention in cross-GPU stealing, we discuss four GPU-only designs: one using peer-to-peer DMA and three relying on host memory. We conclude with a motivation for our preferred host memory-based approach.

Using host memory instead of peer-to-peer DMA for inter-GPU communication allows for a comparative evaluation of performance between host memory-based and direct GPU-to-GPU strategies. Additionally, when using host memory, data management is simpler, as all data required by tasks initially resides in DRAM. Even if the queues are located in host memory, the GPUs can still leverage DMA; thus, accessing the memory directly without CPU involvement is still possible. Communications via host memory can also be used to facilitate load balancing between AMD and NVIDIA GPUs, which is, to our knowledge, not possible using peer-to-peer DMA. The main disadvantage of host memory-centric approaches is that all shared data, such as task queues, must be located in host memory, and thus accessing it incurs PCIe latency and bandwidth limitations. In a peer-to-peer DMA setting, only accesses to shared data that is located on other GPUs suffer the higher PCIe latency and lower bandwidth compared to global memory accesses. Furthermore, if peer-to-peer DMA is achieved through NVIDIA's NVLink [59] or AMD's External Global Memory Interconnect (xGMI) [9], the PCIe bus can be circumvented altogether in favor of the lower latency and higher bandwidth of NVLink and xGMI. For reference, PCIe 4 x16 offers a maximum bandwidth of 31.5 GB/s, whereas the fifth generation of NVLink reaches up to 1,800 GB/s and xGMI up to 448 GB/s [9,59,61].

Peer-to-Peer DMA: In this approach, public queues are stored in global memory and local queues in shared memory. With peer-to-peer DMA, GPUs can access the global memory of other GPUs. Hence, GPU workers can directly steal from workers on other GPUs; see Figure 4. In this approach, random victim selection can be used. However, it can be beneficial to prioritize intra-

GPU stealing over inter-GPU stealing, since stealing from a worker on the same device only requires local device accesses. Prioritizing local workers can be achieved by biasing the victim selection to favor workers located on the same GPU, which we describe further in Section 5.5.

Host-Memory Public Queues: This is the host memory approach we chose for MGWS; in this approach, the public queues are relocated from GPU global memory to host memory, as shown in Figure 5. Storing the queues in host memory enables uniform access for all GPUs, making inter-GPU task stealing as expensive as intra-GPU stealing, since all public queues reside in host memory accessible by all devices. Thus making the design simpler and also helping to distribute the work evenly across GPUs, since inter-GPU work stealing is not favored over intra-GPU stealing.

A potential drawback of this approach is that acquire, release, and steal operations require host memory accesses. However, this is also true for the following host memory-based approaches. While they do not need to transfer tasks as frequently between host and global memory, state management still requires regular host memory accesses.

Host Memory with Reservation Flags: This approach maintains a global task pool in host memory, from which each GPU may claim a partition. Public queues are located in global memory, and local queues are located in shared memory.

Stealing tasks that have already been enqueued on other GPUs poses a challenge. Without peer-to-peer DMA, it is not possible to directly remove tasks from queues residing on remote GPUs. Consequently, this approach indicates stolen tasks through flags associated with the task in host memory. However, this negatively impacts intra-GPU stealing, since public queues may contain a high proportion of already stolen tasks, which reduces the efficiency of steal operations within the same GPU.

To ensure that all tasks in the local queue cannot be stolen, a reserve flag is used, which blocks other GPUs from stealing a specific task. The flag is set when acquiring a task and is cleared when releasing a task. A disadvantage of this approach is that release, acquire, and steal operations require host memory accesses for every task since reservation flags must be updated and checked.

Host memory with MOESI Model: Alternatively, we can model task ownership similar to a cache consistency problem. Where a cache line represents a task and every worker needs a consistent view on that task, i.e., who owns a task, who tries to steal a task, and whether a worker is already executing a task. To that end, each task is assigned a state akin to the MOESI protocol:

- **Invalid**: Task unclaimed
- **Exclusive**: Single GPU owned, not stealable
- **Owned**: Owner has priority; others may steal
- **Shared**: Multiple GPUs may execute; owner still preferred
- **Modified**: Task in execution

Unlike MOESI, a task cannot leave the modified state. Additionally, MOESI relies on snooping for several state transitions. Since GPUs cannot snoop host memory accesses, we must reimplement this with GPU-to-GPU message queues or polling. While this approach allows more fine-grained control over task ownership compared to the previous approach, it is also more complex. Furthermore, state management requires regular access to host memory.

Host-Memory Conclusion: Of the three host memory-based designs, we prefer the public queues in host memory. It simplifies the stealing process, avoids stealing tasks that have already been stolen, and we assume that it does not introduce a significant increase in host memory traffic compared to the other host memory-based approaches.

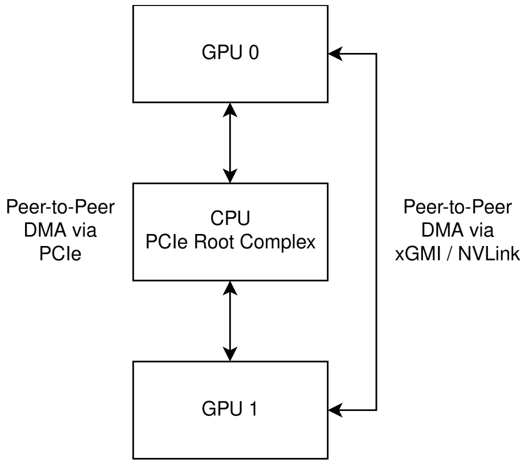

Figure 4 : The GPUs can access each other directly via peer-to-peer DMA either over PCIe or bypass the CPU completely using NVLink or xGMI.

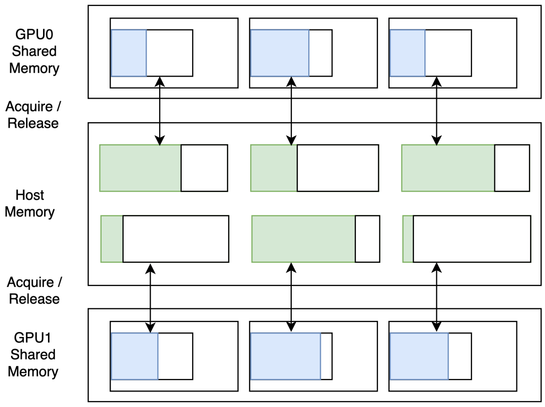

Figure 5 : Multi-GPU work stealing using task queues in host memory. The local portions in blue are located in GPU shared memory. Whereas the public portions in green from which workers can steal tasks reside in host memory. Acquire and release operations require data transfers between shared and host memory.

4.8. Data Management

Efficiently managing the data required by tasks is challenging. While it is possible to use Unified Virtual Addressing (UVA) across multiple GPUs, this is usually not optimal, as UVA relies on page faults to migrate pages between GPUs. Hence, tasks running on different GPUs trying to access the same data might generate a significant amount of page faults, harming performance.

It is also possible to store all task-specific data in registered host memory, making it visible to all GPUs. While this approach will not incur any page faults, the higher latency and lower bandwidth when accessing host memory compared to global memory will likely cause overhead.

A more sophisticated approach is to track the access type, size, and location of data needed by tasks and then manually map or replicate the data onto the GPU that requires the data. Inter-GPU data migration likely also involves heuristics to determine if a specific steal operation or migration of data is beneficial, considering the additional cost of data transfers. This, however, is not the focus of this thesis, and we instead assume that tasks use UVA, registered memory, or do not require additional data.

5. Implementation

This chapter presents the implementation details of MGWS, our work stealing system. We begin by outlining our work stealing algorithm and showing how it takes advantage of key architectural features of modern GPUs. From there, we explain how tasks are represented and how a set of macros simplifies the definition of custom tasks. Next, we describe the layout of orchestration data shared among GPUs and the persistent kernel execution. We also describe randomized target selection and show how a slight bias can reduce the overhead of inter-GPU communications. Two termination detection algorithms follow, including an atomic counter-based approach. Atomic counter-based termination detection is usually inferior to tree-based approaches, but due to highly optimized GPU atomics, it suits the GPU well [49]. We then examine how to determine the maximum number of concurrent workers that can be scheduled on a given GPU. Thereafter, we show how CPU workers are integrated and how a single code base can serve workers across hosts, devices, and device vendors. The chapter concludes with details about our peer-to-peer DMA implementation and how we keep memory accesses on-device when stealing from local workers.

5.1. Work Stealing Algorithm

In this chapter we present our work stealing algorithm. Work stealing is a dynamic load balancing mechanism where workers obtain new tasks by stealing from other workers. Thus providing two main advantages:

1. Tasks are assigned at runtime depending on compute load. As a result, a task-to-worker mapping does not need to be created in advance, which is particularly difficult if task execution times are not known.
2. The algorithm is decentralized; as such, no single worker or task arbiter may become a bottleneck.

In our work stealing algorithm, each worker owns two queues, a local and a public queue. The public queue is larger than the local queue, and its tasks can be stolen by other workers. In contrast, the local queue, which can only be accessed by its owner, contains tasks that will be executed next and allows inserting new tasks. Both queues are implemented as circular buffers, and in the current setup, the local queue can hold 32 tasks, whereas the global queue can store 256 tasks. We choose these numbers to strike a balance between memory usage, load balance, and performance; however, our evaluation in Section 6.7 shows that depending on the inter-GPU communication mechanism, smaller queues can deliver better performance. Bigger queues require fewer transfers of tasks at the cost of consuming more memory. Specifically, large local queues potentially also increase imbalance because more tasks might be stored in local queues and are thus off-limits to other workers.

Public queues can reside in global memory or registered host memory depending on the intra-GPU communication mechanism. Multiple workers can also share the same public queue, allowing for less memory usage at the cost of higher contention. When using peer-to-peer DMA, the public queues are stored in global memory. Thus allowing a worker to access its queue and queues from other workers located on the same GPU without leaving the device. Steal attempts on workers residing on remote GPUs are still possible because public queues are mapped into the virtual address space of all participating GPUs. In the absence of peer-to-peer DMA, the public queues reside in registered host memory. Hence, all accesses to public queues need to leave the device.

The local queue is always located in shared memory. Thus enabling fast access to the tasks that will be executed next but also limiting the size, since shared memory is significantly smaller than global memory. The possible queue layouts can be seen in Figure 1 and 5.

Initial tasks can be stored in host or global memory and are loaded by each worker independently. An atomic counter ensures that every task is only loaded once.

Every public queue has a 64-bit atomic integer as metadata, depicted in Figure 3, which consists of the number of steal attempts in the current epoch, a lock, overflow bits for each epoch, the epoch index, the number of initial tasks in the epoch, and the epoch tail. The lock can only be held by the owner of the public queue and prevents steals while updating the queue's metadata. A public queue also possesses two counters per epoch, one counting the tasks that have been stolen successfully in the epoch and the other tracking the number of steals that began during the epoch. The counter tracking the number of steals in the epoch is set by the queue owner at the end of the epoch because the value is not needed before that point. Letting the queue owner update the counter also helps to avoid potential race conditions caused by the thief not being able to increment the attempted steals field in the 64-bit queue metadata and the counter in the epoch simultaneously. Using these counters, the owner can determine whether a previous epoch has any outstanding steals.

MGWS has the same four main operations as SWS: steal, release, acquire, and execute. We use the remainder of this section to describe our implementation of these operations.

Steal: When a worker runs out of work, it tries to steal new tasks from another worker. It does this by first selecting a target. A worker checks whether a target still has work and claims a portion of the tasks using a single atomic fetch-and-add on the *attempted steals* field of the target's public queue metadata. Every successful steal attempt steals half of the remaining tasks. Thus, the current steal attempt and the initial number of tasks allow the worker to calculate how many tasks have been stolen so far and, if any, how many tasks it must steal. The worker locates the tasks by adding the number of previously stolen tasks to the epoch tail. When copying tasks in any of the four operations, we use all threads of the worker and memory coalescing to maximize transfer speeds. Once all tasks have been moved, the stealing worker first notifies the termination detection mechanism that it has regained work. It then begins a new epoch for its public queue. Finally, the worker marks the steal as completed by adding the number of stolen tasks to the `completed_steals` counter of the epoch in which the steal began.

When letting multiple workers operate on the same data, we must consider memory coherency and consistency. This is especially true on GPUs, which use weakly ordered memory consistency models [47]. To nevertheless ensure memory coherency and a consistent order of the operations, we employ atomic operations and fences.

Release: A worker transfers tasks from the local queue to the public queue when the local queue becomes full or the public queue is empty. First, the worker acquires the lock of the public queue. In case it is the sole owner of the public queue, this is done via an atomic OR on the lock bit of the queue's metadata; otherwise, the worker potentially needs to spin until the other owner unlocks the queue. Now the owner is the only worker allowed to access the queue. Other workers may still increment the attempted-steals counter in the metadata. To avoid race conditions and reduce the number of atomic accesses, the locking bit is only checked afterward;

if it is set, the steal must then be aborted. The owner then transfers half of the tasks from the local queue to the public queue. In the end, the owner initiates a new epoch.

Acquire: When the local queue runs out of tasks but the public queue is nonempty, a worker transfers tasks from its public queue to its local queue, which is done similarly to the release operation. First, the worker acquires the lock of its public queue. Then, it transfers $\min\left(\left\lceil\frac{\text{public queue length}}{2}\right\rceil, \frac{\text{local queue size}}{2}\right)$ tasks from the public to the local queue. A worker only fills the local queue at most halfway because tasks can enqueue new tasks into the local queue. Lastly, the worker transitions its public queue into the next epoch.

Execute: To execute a task, a worker first dequeues the task from its local queue. If this fails because the local queue is empty, it may opt to acquire, steal, or load new tasks from the initial task set. It then starts executing the task. During execution, a worker can enqueue new tasks into its local queue; a worker may also release tasks if the local queue does not have enough room for new tasks.

5.2. Task Struct

The task struct needs all the information required to execute a task. Listing 1 presents the `TaskStruct` and related data types. Every `TaskStruct` contains a buffer for storing a `TaskFunctor` and an array of `DataDescriptor` elements.

Each `DataDescriptor` points to a memory area accessed by the task and specifies the type of access, i.e., read or write. A descriptor may also reference a memory area containing additional descriptors in case the task requires more descriptors than fit directly inside the `TaskStruct`. These descriptors are marked with `AccessRights::Indirection`.

The `functor_buffer` stores an arbitrary `TaskFunctor`, which must satisfy the size and alignment requirements of `TaskFunctor<void>`. These requirements are enforced at compile time via `static_assert`. We cannot store a templated `TaskFunctor` directly as a member because workers need to copy the `TaskStruct` between CPU and GPU, thereby losing all type information, including the vtable. Instead, we store the functor in an untyped buffer and distinguish between functors using their `TaskTag`.

Every task functor is assigned a unique `TaskTag` by deriving from `TaskFunctorBase`. New task functors may also inherit from `TaskFunctor` instead of `TaskFunctorBase`, allowing parameters to be supplied either as raw data or via a custom parameter struct passed through the template. Every task functor must implement a `run` function that takes an `AllLaneHandle` and a `SplitTaskQueue &` as its parameters. The `AllLaneHandle` is part of our framework and represents a contract that ensures that all threads belonging to the worker enter the function. Tasks may enqueue new tasks at runtime by using the `SplitTaskQueue`.

Listing 2 illustrates how a worker executes a task functor. Since the `TaskStruct` does not reveal the specific task functor it contains, the worker performs dynamic dispatch based on the functor's unique `TaskTag`. To access the tag, a worker downcasts the functor to its base type `TaskFunctorBase`. Next, the worker aims to invoke the functor's `run` function; to do so, the worker upcasts the functor to the appropriate `TaskFunctorBase` derivative.

The addition of a new task requires three steps:

1. **Implement the task functor**: A new task is defined by creating a custom task functor. The functor must be derived from `TaskFunctor<T>`, where `T` may be a user-defined struct containing the function parameters. Alternatively, the template parameter can also be left blank if a raw byte array spanning the parameter area is sufficient. The constructor of

```cpp
enum TaskTag : uint64_t { /* Every task has a unique tag */ };
enum AccessRights : uint8_t { ReadOnly, ReadWrite, Indirection };

struct TaskFunctorBase {
    TaskTag tag;

    __device__ __host__ explicit TaskFunctorBase(TaskTag tag) : tag {tag} {}
};

template<typename T = void>
struct TaskFunctor : TaskFunctorBase {
    union {
        alignas(PARAMETER_ALIGNMENT)
          std::array<std::byte, PARAMETER_AREA_SIZE> parameters;
        T named;
    };
};

template<>
struct TaskFunctor<void> : TaskFunctorBase {
    alignas(PARAMETER_ALIGNMENT)
      std::array<std::byte, PARAMETER_AREA_SIZE> parameters;

    __device__ __host__ void run(AllLaneHandle, SplitTaskQueue &) {
        // task entry point
    }
};

struct DataDescriptor { std::byte *data_ptr; uint64_t size; AccessRights rights; };

struct TaskStruct {
    alignas(TaskFunctor<void>)
      std::array<std::byte, sizeof(TaskFunctor<void>)> functor_buffer;
    std::array<DataDescriptor, NUM_REFERENCES> data_descriptors;
    uint64_t num_data_descriptors;

    __host__ __device__ void run(AllLaneHandle , SplitTaskQueue &);
};
```

Listing 1 : The `TaskStruct` and related data structures. Every custom task functor is derived from `TaskFunctor` and can define its own parameters. Custom task functors must satisfy the size and alignment requirements of `TaskFunctor<void>`. These requirements are enforced via `static_assert`, which are omitted from the listing. Any data referenced by the task functor must be specified using the `data_descriptors`.

   `TaskFunctor<T>` takes a unique `TaskTag` and the parameters either as an instance of `T` or as a byte array. The label of the `TaskTag` can be chosen freely, provided the label is not conflicting with other tags. The functor's logic is implemented in the member function `__device__ __host__ void run(AllLaneHandle, SplitTaskQueue &queue)`, which is invoked by a worker when the task is executed. Within this function, a worker can enqueue new tasks via the `queue`, and access the typed parameters via the member `named` or raw parameters via the member `parameters`.
2. **Register the task functor**: The new functor must be registered to MGWS by adding a new entry to the `CUSTOM_TASK_TYPES` macro in `include/task_struct/custom_functor_register.hpp`. The first value of the entry is the label of the unique `TaskTag` selected in the first step, and the second is the type of the newly defined task functor. The macro is used to generate code such as the functor dynamic dispatch based on the `TaskTag`.
3. **Include the task functor**: To include the new functor, an include directive for the file containing the new task functor must be added to `include/task_struct/custom_functor_includes.hpp`.

An example of how to create a new task functor can be seen in Listing 3.

```cpp
template<typename Return = void, typename T>
__host__ __device__ auto run(AllLaneHandle handle, T &el, SplitTaskQueue &task_queue)
  -> std::enable_if_t<std::is_base_of_v<TaskFunctorBase, std::decay_t<T>>, Return> {
  decltype(auto) downcast = static_cast<TaskFunctorBase &>(el);
  switch (downcast.tag) {
    case TaskTag::MEMSET: {
      decltype(auto) task = reinterpret_cast<MemsetFunctor &>(el);
      task.run(handle, task_queue);
      return;
    }
    case TaskTag::CONTAINS_WORD: {
      decltype(auto) task = reinterpret_cast<ContainsWordFunctor &>(el);
      task.run(handle, task_queue);
      return;
    }
    /// ...
    default: {
      assert(false);
    };
  }
}
```

Listing 2 : The function performs dynamic dispatch for task functors based on their task tag. First, the functor is downcast to the base type to access the tag. Next, it is upcast to the appropriate `TaskFunctor` implementation according to the tag, and the `run` function is called. We employ macros to generate this code for every `TaskTag`.

```
/* File: include/task_struct/custom_functor_register.hpp */
#define CUSTOM_TASK_TYPES \
  X(COUNT, CountFunctor)

/* File: include/task_struct/custom_functor_includes.hpp */
#include <task_struct/custom_task_functors.hpp>

/* File: include/task_struct/custom_task_functors.hpp */
struct CountParameters {
    uint64_t *target;
};
struct CountFunctor : TaskFunctor<CountParameters> {
    __device__ __host__ explicit CountFunctor(uint64_t *target) :
            TaskFunctor<CountParameters>(TaskTag::COUNT, CountParameters{.target = target}) {}

    __device__ __host__ void run(AllLaneHandle, SplitTaskQueue &) {
        if (cross::local_id() == 0) { cross::atomicAdd(named.target, 1LU); }
    }
};
```

Listing 3 : Demonstrates how to create a custom task functor. It is necessary to register, include, and implement the new task functor. The `cross` namespace offers wrappers around device-only functions, allowing the `run` method to be executed on host and device.

Area	**Purpose**
TASK	Holds the initial task set
PRIVATE_TASK	Private memory for every device
SHARED_WORKER_INFORMATION	Global metadata shared among workers
PUBLIC_TASK_QUEUE_STORAGE	Storage for public task queues
PUBLIC_TASK_QUEUE_REGISTRY	A Registers storing pointers to all public task queues
GLOBAL_TERMINATION_TREE_STORAGE	Storage for the global termination detection tree
LOCAL_TERMINATION_TREE_STORAGE	Storage for the device-local termination detection tree

Table 5 : The orchestration memory areas and their purpose.

5.3. Orchestration Data

The orchestration data contains global metadata, public queues, and the termination detection mechanism. MGWS distinguishes between several orchestration data areas shown in Table 5.

Throughout MGWS, we use `TypedOffsetPointer` instead of raw pointers. A `TypedOffsetPointer` contains the memory area and type as template parameters and stores an offset into that area, which lets us assign a different base pointer to each area on every device. As a result, we can replicate areas such as TASK or PUBLIC_TASK_QUEUE_REGISTRY on every GPU while keeping the same `TypedOffsetPointer` values. Using offsets is also useful when using GPUs from different vendors, which in our implementation do not share a virtual unified address space. We distribute the base pointers via GPU constants to all workers. GPU constants may differ per device, are set by the host at startup, and can be read by all threads on the device. CPU workers use global variables instead.

The shared worker information, located in its own memory area and shown in Listing 4, contains global metadata used by all workers. Including the global termination detection data structure and how many initial tasks have been loaded.

The public queue registry also has its own memory area and manages the pointers to all public task queues. It initially also contained a bitmap indicating whether a queue might be empty. The bitmap is part of a mechanism to avoid overflowing a queue's attempted-steal counter. However, we later observed that storing the bitmap in global memory to share it between workers incurs significant overhead, making this approach worse than simply checking the attempted-steal counter before incrementing it. Consequently, we moved the bitmap into shared memory, which is possible because the bitmap does not need to be shared between workers. Doing so provides two advantages: reduced contention since the bitmap is no longer shared across workers and faster accesses, as shared memory is significantly faster than global memory. The original motivation for storing the bitmap as part of the registry was the intention to size it dynamically based on the total number of workers. While dynamically sized shared memory allocations at runtime are possible, they are more involved than allocating global memory. Moreover, using dynamically sized shared memory makes the number of workers that can be executed per compute unit dependent on the total number of workers. Previously this dependency did not exist, as MGWS does not use dynamically sized shared memory in any other component. In its current form, MGWS requires the user to specify an upper bound for the number of workers at compile time; future work may explore the use of dynamically allocated shared memory for the bitmap to remove this limitation.

In addition to the global metadata, each CPU worker and each GPU kernel maintains its own worker-specific information. The worker information contains a seed for the random generator used during target selection, a range of all worker IDs residing on the same device, and, when enabled, the device-local termination detection tree.

5.4. Persistent Kernel

A GPU worker is represented by a block inside a persistent kernel. At startup the host initializes queues, metadata, the termination detection mechanism, and the initial task set. Subsequently, the host sets GPU constants, which point to orchestration memory areas for each GPU. Finally, the host launches the GPU kernel seen in Listing 5. We launch one kernel per GPU containing all blocks that should run on that GPU. It is also possible to launch multiple kernels per GPU or even one kernel per worker, but this does not provide any advantages, so we opt to launch a single kernel per GPU. Every kernel receives a continuous range of worker IDs of all workers located on that device, a seed for the random generator, and, when enabled, the device-local termination detection tree. In contrast to the tree-based termination detection, the atomic-counter mechanism has no device-local component and is stored entirely in the global metadata.

Each work can obtain its worker ID by adding its block ID to the lower bound of the ID range. To generate random numbers on the GPU, we use `hiprand` on AMD and `curand` on NVIDIA. These random number generators take not only a seed but also a sequence index, where each sequence generates different random numbers for the same seed. We use a common seed per kernel and the block ID as the sequence index, ensuring that every worker generates a different set of random numbers.

```
template<typename TD>
struct SharedWorkerInformationBase {
    /// the termination detection mechanism
    TD termination_detection;
    /// storage for initial tasks
    TypedOffsetPointer<TaskStruct, TASK_BASE> task_storage;
    /// total number of initial tasks
    uint64_t number_of_initial_tasks = 0;
    /// total number of workers on all devices combined
    uint64_t total_number_of_workers = 0;
    /// number of tasks loaded from the initial task set
    uint64_t initial_tasks_loaded = 0;
    /// number of started tasks
    uint64_t started_tasks = 0;
    /// number of completed tasks
    uint64_t completed_tasks = 0;
    /// used to determine the tree id of a worker when using external storage
    uint64_t used_global_tree_ids = 0;
};
```

Listing 4 : The global metadata structure is shared by all workers. Tracking `started_tasks` and `completed_tasks` is optional and can be disabled through a macro to reduce contention.

5.5. Target Selection

In general, a worker chooses a target to steal from at random, which is optimal according to Blumofe et al. [22]. That said, steal attempts using peer-to-peer DMA are faster when the target is on the same GPU because public queues are stored on the same GPU as their owners. Hence, prioritizing workers located on the same GPU may be beneficial. However, completely suppressing steals to workers on remote GPUs or the host can lead to imbalance across devices. To prioritize workers on the same GPU, the worker first draws a uniformly distributed random number x in the range [0, 1). If x is less than or equal to the constant `OWN_GPU_BIAS`, it selects a target from the same GPU. Next, the worker draws another random number to select the actual target. Using two random numbers gives us fine-grained control over the bias towards stealing from the same device. Listing 6 shows how a worker selects a target.

```
template<TerminationDetection TD>
__host__ __device__ void run_worker(WorkerInformation<TD> worker_information) {
    // Init RandomGenerator, task_queue, public_queue, and local_queue

    // main run loop
    while (!worker_information.should_terminate()) {
        auto task = local_queue.dequeue();

        // does nothing if the local queue is also empty
        if (public_queue.is_empty()) { task_queue.release(); }

        if (task.has_value()) {
            // run task
            worker_information.regained_work();
            task.value()->run(task_queue);
        } else if (!public_queue.is_empty()) {
            // get new tasks from public_queue
            task_queue.acquire();
        } else {
            // public_queue and local_queue are empty
            // if possible load more task from the initial task set
            if (worker_information.load_tasks(task_queue)) {
                continue;
            }

            // steal tasks
            auto target_queue = /* get random target queue */;
            auto success = target_queue.steal_to(task_queue);
            if (success) { continue; }
            worker_information.out_of_work();
        }
    }
    task_queue.destroy();
}
```

Listing 5 : A coarse-grained overview of the run loop executed by each worker. `WorkerInformation` stores metadata such as the worker ID and a reference to the termination detection mechanism. Some implementation details and all thread synchronizations are omitted for readability. The worker first starts by initializing the public and local queues. The `SplitTaskQueue task_queue` struct has access to both queues and facilitates operations involving both. Inside the loop, the worker attempts to dequeue a new task from the local queue. Next, the worker checks whether the public queue is empty; if this is the case, the worker tries to move tasks from the local queue into it. Subsequently, the worker executes the task if one is available; otherwise, the worker tries to acquire new tasks from the public queue. If both queues are empty, the worker attempts to load tasks from the initial task set or to steal tasks from another worker.

```
__device__ __host__ inline uint64_t get_random_worker_id(
  WorkerIDRange global_id_space, WorkerIDRange own_device, RandomGenerator &rng) {
#if CROSS_GPU_STEALING
#ifndef NO_OWN_GPU_BIAS
  if (global_id_space == own_device) {
      auto target = rng.uniform_corrected();
      return own_device.get_id_at(target);
  }
  auto steal_from_own_device = rng.uniform_corrected();
  if (steal_from_own_device <= OWN_GPU_BIAS) {
      auto target = rng.uniform_corrected();
      return own_device.get_id_at(target);
  }
  auto target    = rng.uniform_corrected();
  auto target_id = static_cast<uint64_t>(
      target * static_cast<double>(global_id_space.get_len() - own_device.get_len()));
  if (target_id < own_device.start) { return target_id; }
  return target_id + own_device.get_len();
#else // NO_OWN_GPU_BIAS
  (void)own_device;
  auto target = rng.uniform_corrected();
  return global_id_space.get_id_at(target);
#endif // NO_OWN_GPU_BIAS
#else // CROSS_GPU_STEALING
  (void)global_id_space;
  auto target = rng.uniform_corrected();
  return own_device.get_id_at(target);
#endif // CROSS_GPU_STEALING
}
```

Listing 6 : The function selects a target worker to steal from. Workers residing on the same GPU can be prioritized using the `OWN_GPU_BIAS` macro, and cross-GPU stealing can be disabled with `CROSS_GPU_STEALING`. The `uniform_corrected` draws a random number in the range $[0, 1)$. On single-GPU systems, the worker only needs to draw a single random number. On multi-GPU systems, with a bias for stealing from the same GPU, the worker draws two random numbers; the first determines whether to steal from the same GPU, and the second selects the actual target.

5.6. Termination Detection

Work stealing is a decentralized load balancing algorithm. Each worker knows only whether it still has work but has no global view of whether there is still work in the system.
Without a termination detection mechanism, workers would attempt to steal tasks indefinitely because none of them can determine whether all tasks have already been executed.

This chapter discusses two termination detection algorithms. The first utilizes an atomic counter. The second is based on a binary tree and is inspired by the mechanism used in SWS [23]. Even though the tree-based algorithm has a better asymptotic worst-case complexity of $O(\log n)$ compared to $O(n)$ for the atomic counter approach, where n is the number of workers, in practice GPU atomics are heavily optimized [49]. We therefore want to evaluate whether the simpler atomic-counter method could outperform the more sophisticated tree-based algorithm.

5.6.1. Atomic Counter Termination Detection

In this termination detection mechanism, we use an atomic counter located either in global memory or host memory, depending on the inter-GPU communication mechanism, to track the number of idle workers. An idle worker increments the counter once and later decrements it again if it regains work. As soon as the counter reaches the total number of workers, all workers can safely exit. The worst-case complexity is $O(n)$, where n is the number of workers, which occurs when all workers try to increment the counter simultaneously. Naively incrementing the counter can cause a race condition, which may lead to premature termination. To prevent this race, we introduce a new rule: A worker is only allowed to increase the counter if no steals involving its public queue are still in progress. We discuss this race more and provide an example in Section 5.6.3.

5.6.2. Tree-Based Termination Detection

The tree-based termination detection mechanism utilizes one or more binary trees in which every worker corresponds to a node. Each worker passes a vote up the tree that indicates whether it still has work or has become idle. Inner nodes combine their vote with those of their children. A node reports *out-of-work* only if it and both children have no remaining work. Eventually, all votes are reduced to a single vote at the root. Once the root determines the result, it broadcasts the collective vote back down the tree. If a worker receives an out-of-work signal via this broadcast, it exits, which only happens when every worker in the system reports that it has run out of work. The gathering of votes is equivalent to a binary-tree reduction with complexity $O(\log n)$. The subsequent broadcast also has a complexity of $O(\log n)$, yielding an overall complexity of $O(\log n)$, where n is the number of workers. However, voting out-of-work as soon as the local and public queues become empty can lead to premature termination, as discussed in Section 5.6.3. To address this, a worker is allowed to vote out-of-work only when it has not been targeted by any steals in the current round of voting.

Beyond correctness, we also optimize the mechanism by utilizing multiple binary trees, i.e., a global tree located in host memory and one tree per GPU stored in global memory, as depicted in Figure 6. Here the root of every local tree also participates as a node in the global tree, passing on any votes from the global tree to the local tree and vice versa. Thus, only one worker, the root, needs to access host memory, while the others can cast their votes in the local tree located in global memory with lower overheads.

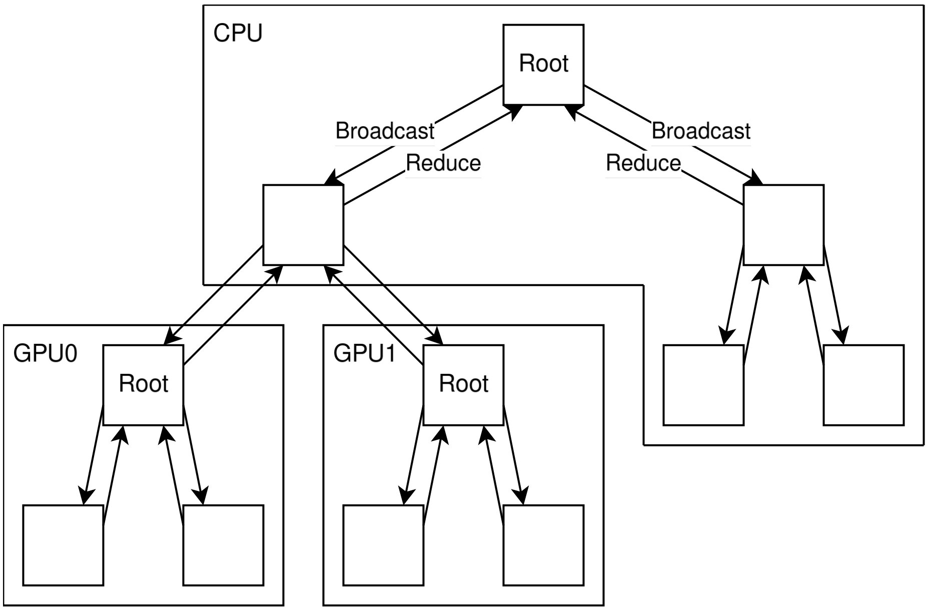

Figure 6 : Illustration of a termination-detection tree with 11 workers: five CPU workers and three GPU workers per GPU. In this example, the root of the global tree is a CPU worker, but a GPU worker could also assume this role. The per-GPU roots belong both to their local trees and to the global tree.

5.6.3. Race Conditions

This section presents and discusses two race conditions that can occur in naive implementations of the two termination detection algorithms. We start with a race condition of the atomic-counter-based mechanism, followed by the tree-based approach. Thereafter, we discuss the differences and similarities between the two race conditions and their respective mitigations.

Atomic-counter-based termination detection: The race condition can occur when we allow workers to increment the counter immediately as soon as the worker becomes idle. The race can lead to premature termination and can thus cause tasks to never get executed. We present the race condition in Table 6. The race occurs because while tasks are in transit, it is possible for the stealing worker and the target to both report that they are out of work. As a result, the counter can reach the total number of workers even though there is still work in the system, causing workers to exit prematurely. To prevent this race, we prevent a worker from increasing the counter if any steals involving its public queue are still in progress.

Table 7 depicts the scenario from Table 6 with the new rule in place. The new rule eliminates the race because even if the target runs out of work during the steal, it postpones incrementing the counter until the steal is completed. At which time, the stealing worker has regained work and, thus, decrements the counter, ensuring that the counter remains less than the total number of workers.

Tree-based termination detection: The race condition can occur when a worker votes out-of-work as soon as the local and public queues become empty and can lead to premature termination, as can be seen in the following scenario. Let B be a worker at the bottom of the tree that ran out of work, and let T be a worker at the top of the tree that still has work. R is the root, and all other workers are out of work.

Worker A	Worker B	Counter
	increments counter	n-1
	steals tasks from A	n-1
runs out of work	steal still in progress	n-1
increments counter	steal still in progress	n
exits because counter reaches total number of workers	steal still in progress	n
	regains work and decrements counter	n-1
	signals steal as complete	n-1

Table 6 : This scenario shows how naively incrementing the termination detection atomic counter can lead to premature termination. The example consists of two workers, A and B. A still has work, while every other worker, including B, has run out of work.

Worker A	Worker B	Counter
	increments counter	n-1
	steals tasks from A	n-1
runs out of work	steal still in progress	n-1
does not increment counter	steal still in progress	n-1
	steal still in progress	n-1
	regains work and decrements counter	n-2
	signals steals as complete	n-2
increments counter		n-1

Table 7 : This scenario depicts the same situation found in Table 6. However, in this version, a worker is only allowed to increment the termination detection atomic counter if it is not currently involved in any steals.

In this scenario the following happens in order:

1. B votes that it is out of work.
2. B passes its vote to its parent.
3. B steals from T and regains work.
4. B votes that it regained work, but this is only considered in the next round of voting.
5. T runs out of work.
6. T passes its vote to its parent.
7. R receives the commutative votes from its children, which both signal out-of-work because B's regained-work vote only reaches the root in the next round of voting.
8. R broadcasts out-of-work to all workers.
9. All workers exit, which is wrong because B still has work.

With this in mind, the issue occurs because votes travel up the tree in waves, and the propagation through the tree takes time. If a worker votes that it is out of work and then steals tasks from another worker, two situations are possible. When the target has already voted, everything remains consistent because the target must have voted that it still has work; otherwise, the steal could not have succeeded. By contrast, a steal from a target that has not yet voted can lead to inconsistencies. The target may run out of work before the voting wave reaches it. As a result, the root may mistakenly conclude that all workers are idle because in that particular wave every worker voted that it is idle.

To address this, only a worker that has not been targeted by any steals in the current round of voting is allowed to vote out-of-work. The same scenario with this additional rule looks as follows:

1. B votes that it is out of work.
2. B passes its vote to its parent.
3. B steals from T and regains work.
4. B votes that it regained work, but this is only considered in the next round of voting.
5. T runs out of work but votes that it still has work.
6. T passes its vote to its parent.
7. R receives the cumulative votes from its children from which one signals that there is still work in the system.
8. R broadcasts that there is still work in the system.
9. All workers continue execution.

Adding the new rule alleviates the issue because when a worker who has already voted steals from a target that has not yet voted, the target always votes that it still has work. Subsequently, a new round of voting begins in which the new vote from the stealing worker is considered by the root. Both scenarios are also illustrated in Figure 7.

Comparing the race conditions: Both race conditions can cause premature termination, which may lead to tasks not getting executed. The mitigations for the race condition are also similar, i.e., workers are not allowed to report that they are out of work if a steal targeting them is still in progress. However, while both termination detection mechanisms face the same problem: Tasks in flight are not owned by any worker; the root cause for the tree-based termination

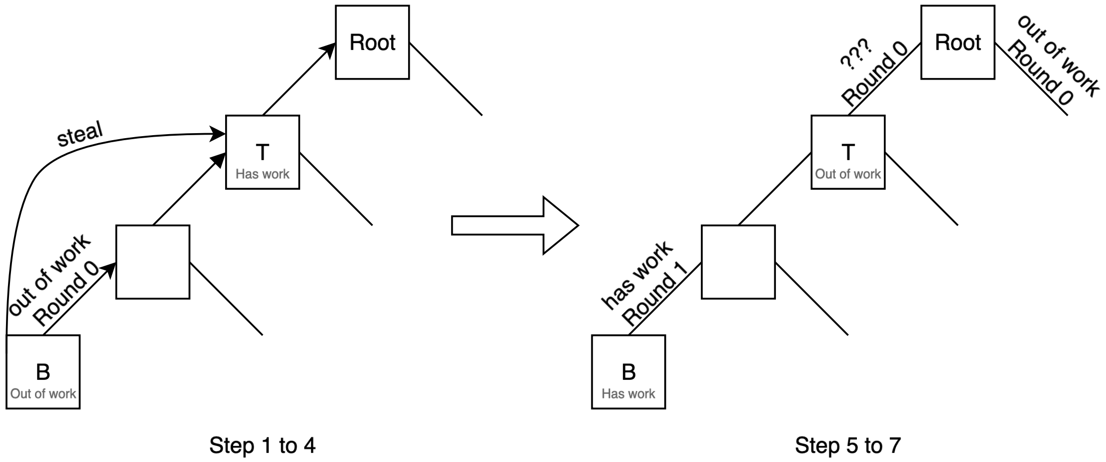

Figure 7 : A visualization of the problematic termination detection tree scenario. The ??? vote depends on whether a worker is forbidden to vote that it is out of work if it was targeted by a steal in this round of voting. If it is forbidden to do so, the vote will be “has work” and another round of voting starts. Otherwise the vote will be “out of work”, and the root will broadcast to all workers that they should exit.

detection is not the tasks in flight but rather the multiple rounds of voting and that therefore a vote takes time to propagate to the root.

5.7. Determine the Number of Workers for Maximum Occupancy

Each worker is represented as a block, and we aim to launch as many workers on the GPUs as can run concurrently. To achieve this, we must determine the number of blocks needed to maximize occupancy, i.e., the maximum number of warps that can execute on a compute unit simultaneously. All warps belonging to a block are always scheduled on the same compute unit, and GPUs do not support preemption. As a result, a block remains on the compute unit where it is launched.

The maximum number of blocks per compute unit depends on the resources consumed by the block, primarily registers and shared memory. Launching too few blocks wastes compute resources, while launching too many blocks causes some workers to never run as there is no preemption. Launching too many blocks interferes with termination detection since we would first need to determine which workers are running. Even if the termination detection algorithms were adapted to tolerate blocks that never launch, surplus blocks would still waste memory since their queues and metadata must still be allocated. Furthermore, compute resources would be wasted because steal attempts directed at those blocks always fail.

Usually every compute unit has its own shared memory and registers. Hence, the desired number of workers needs to be a multiple of the number of compute units. However, GPUs developed using the AMD RDNA architecture group compute units into work group processors (WGP) [5]. Each WGP contains two compute units that share the same shared memory [5]. Thus, on these systems, the maximum number of blocks is a multiple of the number of WGPs.

Functions such as `hipOccupancyMaxPotentialBlockSizeVariableSMem` also provide a lower bound for the number of blocks that can concurrently execute a given GPU kernel. To determine the exact upper limit, we start from this lower bound and increment the number of workers by the number of compute units, or WGPs, until we reach a point where not all workers can be launched concurrently. While the shared memory usage of the work stealing algorithm is independent of the number of warps, the register usage is not. Therefore, we must determine the maximum number of schedulable wraps next. The maximum number of threads per block is hardware limited; for example, the AMD RX 6950 XT supports up to 1024 threads per block.

The compiler already estimates the number of registers and amount of shared memory required by each block and limits the number of blocks per compute unit accordingly. If the kernel makes use of dynamic shared memory, specified at launch time, the number of blocks per compute unit may be reduced even further at runtime. As a result, tasks that require large amounts of shared memory or registers can limit the number of workers that can be executed concurrently. Even if those tasks are never executed, because the compiler must assume a worst-case scenario. For this reason, only tasks that will actually be executed should be compiled.

In our evaluation, depending on workload and hardware configuration, we schedule in the order of a hundred workers distributed over two GPUs and the CPU, one per compute unit and CPU core.

```
namespace cross {
template<typename T>
__host__ __device__ inline T atomicAdd(T *a, T b, std::memory_order order) {
#if (defined(__HIP_PLATFORM_AMD__) && defined(__HIP_DEVICE_COMPILE__))
    switch (order) {
        case std::memory_order_relaxed: return ::atomicAdd_system(a, b);
        /* ... */
        case std::memory_order_seq_cst: {
            __threadfence_system();
            auto ret = ::atomicAdd_system(a, b);
            __threadfence_system();
            return ret;
        }
        default: {
            __threadfence_system();
            auto ret = ::atomicAdd_system(a, b);
            __threadfence_system();
            return ret;
        }
    }
#elif defined(__HIP_PLATFORM_NVIDIA__) && defined(__HIP_DEVICE_COMPILE__)
    static_assert(cuda::atomic<T>::is_always_lock_free);
    return reinterpret_cast<cuda::atomic<T> *>(a)->fetch_add(b, std_to_cuda_memory_order(order));
#else
    static_assert(std::atomic<T>::is_always_lock_free);
    return reinterpret_cast<std::atomic<T> *>(a)->fetch_add(b, order);
#endif
}
}
```

Listing 7 : Unified wrapper for the device function `atomicAdd`. Additional wrappers for other device functions also reside in the `cross` namespace. The wrapper selects the appropriate implementation depending on the compilation target: AMD GPUs use `atomicAdd_system` combined with `__threadfence_system` to enforce memory ordering, NVIDIA GPUs use `cuda::atomic`, and the host uses `std::atomic`. The abstraction provides a consistent interface across all supported platforms.

5.8. Cross-Platform Work Stealing

Our work stealing system enables collaboration between GPU and CPU workers through host memory. CPU workers, in particular, share the same code base as GPU workers, ensuring consistency and maintainability. Given sufficient hardware support, MGWS can theoretically operate across GPUs from different vendors at the same time. However, we were unable to test this scenario because the NVIDIA GPUs we had access to lacked proper PCIe atomics support.

To allow cross-platform code sharing, we use wrappers around device-only functions, allowing separate implementations for each GPU vendor and the CPU; one such wrapper can be seen in Listing 7. MGWS shares parts of its support library with the latest codebase of GPU4FS [50], a GPU file system, particularly the wrapper functions that enable a unified code base on CPU and GPU. Using the same code base for both GPU and CPU workers offers several advantages, including less code to maintain and the ability to use CPU debuggers, which are often more reliable than GPU debuggers.

We also provide dummy implementations for all HIP host functions used by our project, which allows compilation of MGWS without HIP installed. Thus, even without HIP, CPU workers can still run, making it possible to execute automated tests on systems without GPUs, for example, as part of a Continuous Integration (CI) pipeline. For testing, we use the Google Test framework, and we supply wrappers for the Google Test assertions so that the same tests can be executed on both GPU and CPU backends. Even if CPU-only tests cannot detect race conditions that exclusively occur on the GPU, they still provide valuable insights and catch a wide range of errors early.

We used ROCm HIP [4] to implement our work stealing algorithm. While ROCm HIP supports AMD and NVIDIA GPUs, it cannot handle both GPU vendors in the same executable. To nevertheless enable collaboration between NVIDIA and AMD GPUs, we support storing all tasks, public queues, and information needed for termination detection in a shared memory-mapped area. The shared area can then be mapped by an executable for each vendor. Due to Address Space Layout Randomization (ASLR), offsets need to be used instead of raw pointers for accesses into the shared memory area. We are unable to test collaboration between AMD and NVIDIA GPUs because we only had access to consumer NVIDIA cards, on which PCIe atomics targeting the shared memory area were unreliable. Since PCIe atomics are a fundamental requirement for our work stealing algorithm, reliable, cross-vendor testing was not possible in our setup.

5.9. Utilizing Peer-to-Peer DMA

Using peer-to-peer DMA allows GPUs to perform load and store operations on the global memory of other GPUs through PCIe, NVLink, or xGMI. Due to a lack of compatible GPUs, we only test peer-to-peer DMA via the PCIe bus, but MGWS should also support NVLink and xGMI if available because all three options use the same interface. To enable PCIe-based peer-to-peer DMA, it needs to be supported by the motherboard and enabled in the kernel configuration by setting `CONFIG_HSA_AMD_P2P`. The HIP built-in function `hipDeviceCanAccessPeer`, which returns whether a given GPU can access another GPU's memory, can be used to ensure that peer-to-peer DMA is available.

Even if peer-to-peer DMA is enabled, atomics over the peer-to-peer connection are not guaranteed to work as expected and depend on the capabilities of PCIe switches and upstream bridges between the GPUs [11]. To test whether atomic operations are supported, we wrote a test program, which we provide in Listing C10. First, the program enables peer-to-peer DMA, and then one GPU spins until a certain memory location changes, while the other sets that memory location atomically. Thus, simulating a spin lock, if atomic operations work correctly, we would expect the spinning GPU to exit after a short delay. However, running our program on two GPUs connected to non-adjacent PCIe slots, we observe that although data transfers via the peer-to-peer connection succeed, the spinning GPU never stops. Thus suggesting that atomic coherency is not provided in this configuration. We presume that this is because the two PCIe slots used are likely connected to different PCIe root complexes. However, due to the physical constraints of the GPUs, we were unable to verify whether alternative configurations could resolve this issue.

Although our experiments indicate that atomic operations over PCIe-based peer-to-peer DMA are not reliable on our test system, we have fully implemented a peer-to-peer DMA-based design in MGWS. Due to the lack of compatible hardware, we were unable to validate this implementation experimentally. In the following, we describe the changes to MGWS that allow it to take advantage of the peer-to-peer connection in order to minimize off-device accesses.

When using peer-to-peer DMA to facilitate inter-GPU communication, all orchestration data is stored in global memory. We also replicate the public queue registry onto every GPU to avoid peer-to-peer accesses in favor of local memory accesses whenever possible. Some information, such as the shared worker information that contains the global termination mechanism, cannot be replicated and is therefore stored on the GPU with device ID 0, which we designate as the main GPU by default.

The public queues of workers are stored on the same GPU on which they are scheduled. Thus, acquire, release, and steal operations to workers on the same GPU only require access to local global memory. Storing queues in global memory alleviates bottlenecks and overheads encountered when using host memory to store the public queues, since in that case all operations targeting any public queue, including a worker's own public queue, require host off-device accesses to host memory.

6. Evaluation

In this section, we evaluate the performance characteristics of MGWS in various configurations. Most of our evaluation is conducted using three different setups. First, the orchestration data, including public queues and termination detection, reside in global memory. When using global memory, MGWS can only utilize a single GPU; however, this configuration allows us to assess the upper performance limit of MGWS. Second, the orchestration data is stored in host memory. Using host memory enables the use of multiple GPUs and the CPU and represents one of our two inter-GPU communication schemes, where every worker accesses a shared area in host memory. The third aims to approximate the performance of the peer-to-peer DMA-based inter-GPU communication mechanism by storing orchestration data in the global memory of a remote GPU. Storing all public queues in remote global memory is not the same as our planned peer-to-peer DMA-based approach. In contrast, our peer-to-peer DMA-based mechanism stores public queues on the same GPU where their owners reside. We therefore expect the real peer-to-peer DMA-based approach to have higher performance because release, acquire, and local steal operations do not need to leave the device. By storing all public queues in remote global memory, release and acquire operations incur additional overhead, and all steal operations are similar to steal operations targeting a worker residing on the other GPU. However, our evaluation systems do not support coherent atomic operations over the peer-to-peer connection. As a result, the remote global memory setup can only support a single GPU and represents the closest approximation we can achieve to the genuine peer-to-peer DMA-based inter-GPU communication scheme.

The evaluation section is structured as follows: First, we introduce our two test systems in Section 6.1. We primarily use the *server*; the *desktop* is only used in Section 6.10, where we analyze the host memory bottleneck since the desktop allows us to more easily change the DRAM speed. Afterwards, we present the two workloads under evaluation. The *MEMSET* workload has short regular tasks, whereas the *CONTAINS* workload features longer irregular tasks. Next, we discuss the correctness of our setup and MGWS. Then we discuss the potential effects of fully utilizing the GPU with persistent kernels. Thereafter, we investigate the overhead caused by detecting overflows in the steal attempt counter and whether sophisticated overflow prevention is justified. This is followed by a comparison of our two termination-detection algorithms. We then assess the overhead caused by sharing ownership of public queues. Afterwards we discuss single-GPU performance by comparing the three memory configurations, host memory, global memory, and remote global memory, using both the regular and irregular workloads. We also compare MGWS to directly executing the workloads with static load balancing, assigning each worker an equal number of tasks. Following that, we investigate MGWS using multiple GPUs and CPU workers also executing both workloads. Without coherent atomics over the peer-to-peer DMA connection, the multi-GPU evaluation is limited to the host memory-based inter-device communication. Thereafter, we investigate the host memory bottleneck by reducing memory speeds on our secondary test system. Finally, we discuss the overall findings of our evaluation.

All our measurements use 1000 tasks per second as a metric and are reported with error bars showing the minimal, maximal, and average tasks per second. When not specifically stated otherwise, MGWS uses the counter-based termination detection because it performs better than the tree-based approach.

6.1. Test Systems

We evaluate MGWS on two systems referred to as *server* and *desktop*. The desktop is only used to explore the host memory bottleneck, which we discuss in Section 6.10; the rest of our evaluation uses the server. We use two different systems because the server supports peer-to-peer DMA and contains two dedicated GPUs, whereas the desktop only contains a dedicated GPU and integrated GPU but allows us to change the DRAM speed more easily.

Server

- **Processor:** AMD EPYC 9124 with 16 cores @ 3.0GHz
- **Memory:** 64 GB of DDR5 at 4800 MT/s distributed into four 16 GB DIMMs, 12 channel
- **GPU1 RX6950:** AMD Radeon RX 6950 XT with 16 GB of VRAM, 64 KiB of shared memory per block, 40 work group processors (80 CUs), and connected via PCIe 4.0 x16
- **GPU2 RX6600:** AMD Radeon RX 6600 with 8 GB of VRAM, 64 KiB of shared memory per block, 14 work group processors (28 CUs), and connected via PCIe 4.0 x16
- **Software** Fedora Kernel version: 6.18.0, HIP version: 7.1.52802

Desktop

- **Processor:** AMD Ryzen 7 7700X with 8 cores @ 4.5GHz
- **Memory:** 32 GB DDR5 at 5600 MT/s (slower speed 2400 MT/s) distributed into two 16 GB DIMMs, dual channel
- **dGPU:** AMD Radeon RX 6950 XT with 16 GB of VRAM, 64 KiB of shared memory per block, 40 work group processors (80 CUs), and connected via PCIe 4.0 x16
- **iGPU:** integrated into the CPU, uses host memory as global memory, 64 KiB of shared memory per block, and 1 work group processor (2 CUs)
- **Software** Nanjaro Kernel version 6.12, HIP version: 6.2.41134

6.2. Workloads

We evaluate two workloads: The first sets 8 bytes at a given address to a given value; we call this workload *memset*. The second counts how many Wikipedia articles contain a given word; we refer to this workload as *contains*.

The memset workload is regular and has only a short runtime; we use this workload to evaluate the overhead of MGWS and to assess an upper limit on how many tasks can be executed in a given time. All benchmarks involving this workload execute 1 048 576 tasks. We chose this number to strike a balance between the impact of the startup latency and the time it takes to execute the benchmarks. The total runtime is recalculated into tasks per second, which is obtained by executing each experiment five times. In most cases, the variance between runs is limited; thus, five iterations are sufficient.

The contains workload scans the first 100 000 articles of the German Wikipedia dump and determines how many contain the word “zwischen”, which is German for “between”; we selected this word because we wanted to search for a word that occurs in a significant portion of the corpus. The word “zwischen” is the 129th [3] most common word in the German language and appears in 39 830 out of 100 000 articles. We only consider the first 100 000 articles to limit the runtime, which is especially important when only testing a single worker. We again express our results in tasks per second, which is equivalent to the articles scanned per second in this scenario. The CONTAINS workload is irregular since Wikipedia articles have varying lengths, and a worker can stop scanning an article as soon as it finds the search term. Simply counting the occurrences of the search term in the Wikipedia corpus would be a regular workload because the corpus could be evenly divided across all workers. We use the CONTAINS workload to assess whether MGWS can prevent imbalance in irregular workloads and to evaluate performance when a significant amount of the compute time is spent on tasks rather than the work stealing algorithm. As before, the total runtime is recalculated into tasks per second, which is obtained by executing each experiment ten times. We repeat each CONTAINS run twice as often due to higher per-iteration variance; for single- and multi-GPU measurements, we additionally report results obtained using 200 repetitions to further reduce the influence of outliers.

While our evaluation is limited to two workloads, we choose these workloads to investigate two specific areas. First, to analyze the overhead of individual MGWS components and the theoretical maximum performance achievable under different configurations. Second, to evaluate MGWS under irregular workloads and to determine whether the overheads and differences between configurations observed for the regular workloads remain significant when task execution time dominates overall runtime.

Our results indicate that the selected workloads are sufficient to address these areas. Nevertheless, future evaluation may yield additional insights by incorporating a broader range of workloads.

6.3. Correctness

The following analyzes the correctness of MGWS and the evaluation systems. We first highlight several parts of this thesis where we already addressed correctness and potential race conditions. This is followed by an explanation of how correctness is ensured during the execution of the evaluation workload, and finally we examine the correctness of the evaluation systems.

Various parts of the thesis analyze race conditions and their implications for correctness. In particular, we analyze race conditions in the attempted-steal counter overflow detection algorithm introduced in Section 4.3, motivating the implementation of steal damping as proposed by Cartier et al. [23] instead. When public queue metadata is accessed concurrently, race conditions may arise, which we describe in Section 5.1. We present potential race conditions in the termination detection algorithms, along with corresponding mitigation strategies in Section 5.6.3. In Section 5.8, we explain how a shared code base between the CPU and GPU improves maintainability and enables the use of CPU debuggers, which are often more reliable than GPU debuggers. Moreover, the ability to build MGWS without HIP installed makes Continuous Integration (CI) on systems without GPUs possible.

Now we focus on the correctness of our evaluation workloads. Each of the n tasks in the MEMSET workload is assigned a unique number x from 1 to n, and each task writes x at the offset x in a contiguous memory region. This setup allows us to verify after each run whether all tasks have been executed and whether each task has written to the correct location in memory. We also maintain counters to check how many tasks are executed by each worker and the total number of tasks completed. These counters can be used to detect starving workers or provide another way of validating whether the correct number of tasks has been executed. However, we disabled these counters during evaluation to eliminate any potential impact on measurements. Disabling the counters is particularly important for the comparison of the attempted-steal counter overflow prevention in Section 6.5 or the comparison of the termination detection algorithms in Section 6.6, where incrementing a global tasks counter would likely incur more overhead than the mechanisms under evaluation.

The CONTAINS workload always uses the same search phrase and always checks the same set of articles. Therefore, although we did not externally validate the correctness of our calculated results, during evaluation we check after every run that the same number of articles are found. While this does not guarantee the correctness of the workload, it allows us to compare the result to the simplest setup, that being a single CPU worker, thereby increasing our confidence in their results correctness. Furthermore, it is highly unlikely that a race condition or other errors would produce identical results.

Given that the CONTAINS workload measurements experience a high degree of noise in the execution times, we decided to increase the repetitions from 10 to 200. However, during this, we found that using CPU workers is unstable, sometimes leading to the wrong number of matched articles. The instability prevented us from conducting 200 repetitions whenever the CPU was involved. We did not observe this effect during the 10 repetitions or GPU evaluation.

Having described our workloads, we now turn to our evaluation systems. First, we examine the server, which is the only one of our systems that supports peer-to-peer DMA. HIP specifies three scopes for atomic operations: block, device, and system, but although we use system-scoped atomics over the peer-to-peer DMA connection, the operations are not coherent between the two GPUs. Thus preventing us from evaluating our peer-to-peer DMA-based inter-worker communication mechanism directly. We explore potential reasons for this in Section 5.9 and explain how we reached these conclusions using a spin lock-based tester provided in Listing C10. However, the atomic operations targeting remote global memory using the peer-to-peer connection are coherent at the device scope, as confirmed during testing. Consequently, we can approximate the peer-to-peer DMA-based communication approach by storing data on the remote GPU. Adopting MGWS to store data on the other GPU instead of the one performing the computation was straightforward, since we already have a wrapper around `hipMalloc`, the hip function used to allocate global memory, in place. We just added a conditional compilation block that directs the allocation to the other GPU. Thus, all configurations that use remote global memory perform all allocations on the other GPU instead of the computing GPU.

Next, we highlight issues encountered on the server due to an older kernel and HIP version. Before updating to kernel 6.18 and HIP version 7.1.52802, the system was running kernel 6.16 and HIP version 6.1.40093. Both HIP installations ran inside our own Docker image based on the Dockerfile provided in the official rocm-examples GitHub repository [13]. When using the older kernel, calling `hipDeviceCanAccessPeer`, the HIP function used to determine whether peer-to-peer DMA is supported between GPUs, indicated that peer-to-peer DMA was available. However, enabling peer-to-peer DMA via `hipDeviceEnablePeerAccess` resulted in a null pointer dereference within the kernel as soon as global memory was allocated using `hipMalloc`. The null pointer dereference likely occurs during the allocation because once both directional peer access is established, all current and future allocations are mapped to both GPUs. Furthermore, executing our evaluation workloads frequently triggered the kernel watchdog to report a soft lockup. After updating the kernel and HIP version, none of these issues were observed.

On our second evaluation system, the desktop, we encountered issues using the AMD iGPU. Our initial evaluation was conducted using a 6.12 kernel and HIP version 6.2.41134. On this kernel and HIP version, `hipMalloc` consistently returned a null allocation, and synchronization using HIP streams was unreliable. In particular, kernels were sometimes launched before preceding memory operations on the same stream had completed, unless explicit synchronization using `hipStreamSynchronize` was performed before launching the kernel. Furthermore, the iGPU evaluation is limited to a single worker, as in some configurations, particularly using the tree-based termination detection, this represents the upper limit for the concurrent workers supported by the iGPU. Nevertheless, we were successfully able to conduct our evaluation of the varying DRAM speeds using both GPUs. However, after updating the system, thereby also updating the kernel to 6.18 and HIP to 7.1.52802, we observed that the MEMSET workload frequently reported missing tasks when using the iGPU. Additionally, we observed the name of the iGPU changing and `hipMalloc` returning a valid allocation. While downgrading to the older kernel and HIP version caused the reported name of the iGPU to revert, we were unable to fully replicate the previous state as `hipMalloc` kept returning a valid allocation, and the MEMSET workload also kept reporting missing tasks. As such, the measurements presented as part of our investigation into the host memory seen in Section 6.10 are the original measurements taken before updating the system. Using the older measurements, however, does not affect the correctness or representativeness of the arguments presented in Section 6.10, as we are not comparing performance measurements between the two evaluation systems; instead, the analysis consists only of relative differences between configurations within the same system.

In conclusion, we consider our algorithm to be correct. Moreover, because no errors were observed when using the updated kernel and HIP version during our evaluation, our confidence in MGWS has increased significantly for both single- and multi-GPU settings. We believe that most issues encountered with the older kernel and HIP versions can be attributed to driver-related problems. Nevertheless, investigating the remaining instability observed when using CPU workers and the iGPU remains an interesting direction for future work of MGWS, even if iGPUs are not the primary target platform for MGWS.

6.4. Effects of Fully Utilizing the GPU with Persistent Kernels

We determine whether fully utilizing the GPU with persistent kernels running MGWS has any negative effects, such as thermal throttling. To that end, we launch one worker with 512 threads on each compute unit of the RX6950, resulting in a total of 80 workers, which is the maximum number of workers and threads that can run concurrently. We disabled termination detection and let the workers idle for an hour to analyze if any thermal throttling takes place. Without the termination detection algorithm, idling workers will constantly try to steal work from other workers.

Our measurements did not find any evidence of thermal throttling with a consistent clock speed of 2646 MHz and a maximum die temperature of 79 °C while fully utilizing the GPU's compute pipeline. The power consumption also topped off at 203 W, which is 86 W less than the theoretical limit for that specific GPU of 289 W.

We believe this is because modern GPUs contain a high number of special function units, which we do not use, such as media encoders, media decoders, AI accelerators, ray tracing accelerators, and graphics primitive assembly. Thus, even if we fully utilize the compute pipeline, the GPU die contains a high portion of dark silicon [29], which means parts of the chip that are not powered can absorb heat. Dark silicon also explains why the power draw does not get closer to the theoretical limit.

Depending on the workloads and environmental factors, the temperature and energy consumption of MGWS may vary. However, the experiment indicates that thermal throttling is not a factor that needs to be considered in the remainder of our evaluation.

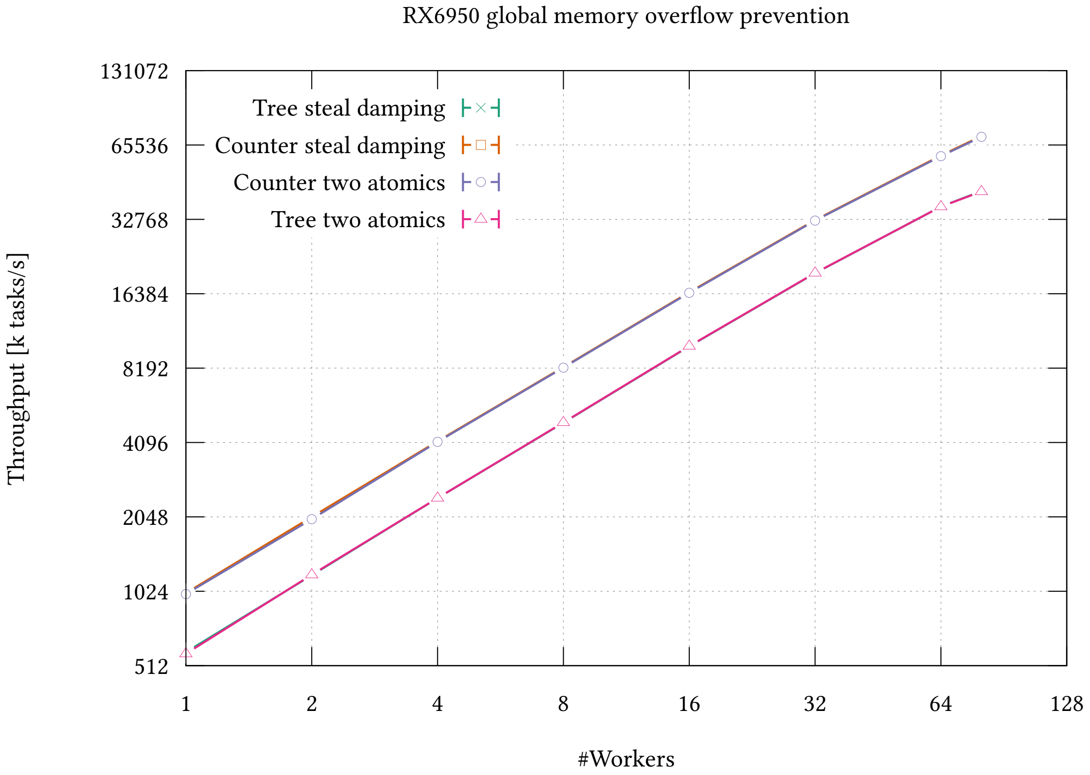

Figure 8 : A comparison of MGWS using two different overflow prevention mechanisms and termination detection algorithms in global memory using the RX6950. The simplest way to avoid the attempted steal counter from overflowing is to perform an atomic fetch before the atomic fetch-and-add to determine whether the victim has any tasks to steal. This approach is compared to the steal damping presented by SWS [23]. The measurements reveal a clear performance difference between the termination detection algorithms. However, the measurements for the two overflow prevention mechanisms are so similar that the *Counter two atomics* plot almost completely overlaps the *Counter steal damping* plot, and the *Tree two atomics* overlaps the *Tree steal damping* plot.

6.5. Overhead of Overflow Prevention

When stealing work from another worker's public queue, the thief must perform an atomic fetch-and-add onto the metadata of the victim's public queue. The increment can lead to an overflow in the attempted steal counter, which must be detected or prevented. The simplest way to avoid this overflow is to perform an atomic fetch before the atomic fetch-and-add to determine whether the victim has any tasks to steal; if not, the attempted steal counter is not incremented. A more sophisticated solution, as presented by SWS [23], uses *steal damping*, which only uses two atomic operations if the previous steal attempt determined that the particular victim has no work left. We intend to assess whether using a sophisticated overflow prevention algorithm is

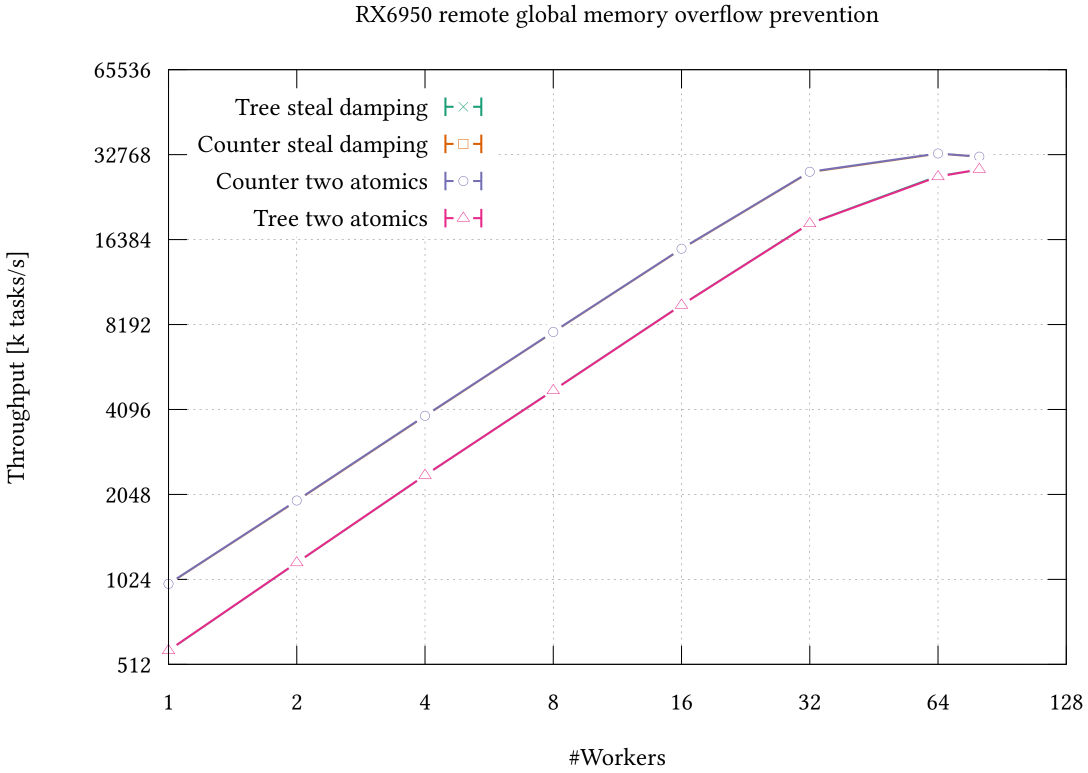

Figure 9 : A comparison of MGWS using two different overflow prevention mechanisms and termination detection algorithms on the RX6950 using the global memory of the RX6600 via peer-to-peer DMA. The simplest way to avoid the attempted steal counter from overflowing is to perform an atomic fetch before the atomic fetch-and-add to determine whether the victim has any tasks to steal. This approach is compared to the steal damping presented by SWS [23]. The measurements reveal a clear performance difference between the termination detection algorithms. However, the measurements for the two overflow prevention mechanisms are so similar that the *Counter two atomics* plot completely overlaps the *Counter steal damping* plot, and the *Tree two atomics* overlaps the *Tree steal damping* plot.

even necessary or if performing the two atomic operations is sufficiently fast. To achieve that, we compare the two overflow prevention approaches with both termination detection mechanisms and use host, global memory, and global memory of the other GPU to store the public queues. The global memory measurements are depicted in Figure 8, the remote global memory results in Figure 9, and the host memory measurements in Figure 10. The figures also provide insights into the performance difference of the termination detection algorithm, which we discuss in Section 6.6.

Our measurements show that using two atomics instead of steal damping is at most 2.11% slower, but in most cases the difference is below 1% in either direction, which applies to the global memory, remote global memory, and host memory setups. Thus, using a sophisticated overflow prevention mechanism is not necessary in our setup. This is surprising, especially when using host memory or remote global memory, because there the access latency is higher compared to global memory, and thus performing two atomic operations that depend on each other should cause greater overheads.

Our results highlight an interesting difference between GPU and CPU work stealing. In their CPU work stealing paper, the authors of SWS [23] emphasize the importance of combining the atomic operations used to check whether a target has work with the operation that actually steals the work into a single access rather than performing two separate accesses. Therefore, suggesting that, at least using distributed memory, such an optimization is important in a CPU work stealing setting, but our observations indicate that this does not translate to the GPU setting.

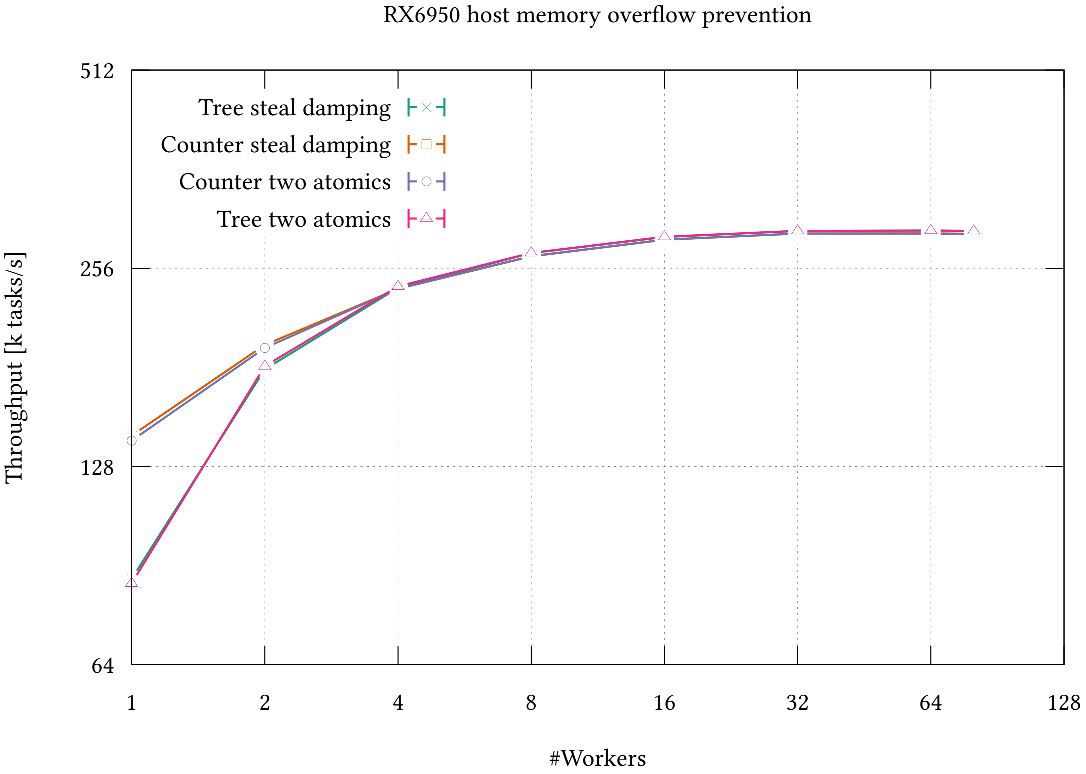

Figure 10 : A comparison of MGWS using two different overflow prevention mechanisms and termination detection algorithms in host memory using the RX6950. The simplest way to avoid the attempted steal counter from overflowing is to perform an atomic fetch before the atomic fetch-and-add to determine whether the victim has any tasks to steal. This approach is compared to the steal damping presented by SWS [23]. The measurements reveal a clear performance difference between the termination detection algorithms until four workers. However, the measurements for the two overflow prevention mechanisms are so similar that the *Counter two atomics* plot almost completely overlaps the *Counter steal damping* plot, and the *Tree two atomics* overlaps the *Tree steal damping* plot.

6.6. Termination Detection Mechanisms

We use the MEMSET workload to compare the overhead of the atomic-counter-based and tree-based termination detection mechanisms. We aim to investigate whether a specific communication scheme is better suited for either of the termination detection algorithms by examining three configurations: executing the workload using a single GPU in global memory, a single GPU in host memory, and a single GPU where the termination detection data is located on the other GPU, necessitating peer-to-peer DMA. We only use a single GPU in all our tests to make them more comparable since, with our current evaluation setup, only the host memory approach can utilize multiple GPUs.

Global memory: The measurements depicted in Figure 8 show that regardless of the number of workers, the atomic-counter-based approach is consistently between 61.02% and 73.54% faster than the tree-based approach. We did not expect the atomic counter to outperform the tree-based approach so clearly, especially with a high number of workers. We suspect this has two reasons: First, GPU atomics are highly optimized; thus, even if this approach has higher contention, it still outperforms the overhead caused by the more complex tree-based approach. Second, the tree-based approach is more complex and thus may still offer more opportunities for optimizations compared to the atomic-counter-based approach. A fully optimized version may shrink the gap to the atomic counter.

Remote global memory: Figure 9 shows a comparison of the termination detection algorithms using global memory located on the other GPU to investigate how they may perform using the peer-to-peer DMA-based communication scheme. Both schemes exhibit a linear speedup until 32 workers, where the counter-based approach is between 34.21% and 41.91% faster than the tree-based approach. Thereafter, the performance gap between the two shrinks, but at 80 workers, the counter-based algorithm is still 10.11% faster. We believe this is due to the same reasons discussed in the global memory setting. However, the counter-based approach seems to be more impacted by the additional latency introduced through peer-to-peer DMA, which reduces its performance lead compared to the global memory setting.

Host memory: Our evaluation in Figure 10 shows that except for the single worker case, the tree-based and counter-based approaches exhibit nearly identical performance. These results are surprising since the tree-based approach uses multiple trees where each GPU has a local termination detection tree in global memory, while only the global termination detection tree resides in host memory. As a result, most accesses target the local trees in global memory, and only a single worker accesses the global tree in host memory. In contrast, the counter-based approach requires all workers to access host memory. Based on this difference we initially expected, the tree-based approach to outperform the counter-based approach. Instead, the counter-based approach merely loses the performance advantages observed in the global memory setting. We hypothesize that the multi-tree optimization is responsible for the removal of the performance difference between the two approaches in host memory. Thus, introducing multiple atomic counters similar to the use of multiple trees could further improve the counter-based approach and potentially lead to it outperforming the tree-based approach in the host memory setup as well. We leave the evaluation of this optimization to future work, as our focus has been on keeping the counter-based approach as simple as possible.

The outlier observed for a single worker is likely due to the absence of a local tree in this configuration, causing all accesses to target the global tree in host memory.

Discussion: The presented measurements show that approaches that might work well in conventional CPU work stealing algorithms, in this case a tree-based termination detection mechanism, do not always translate to the best solution for GPU work stealing, as seen by the performance lead of the counter-based approach in global memory. We also observed that the termination detection algorithms may be differently impacted by the communication mechanism and that a hierarchical solution, especially when using host memory, might be beneficial.

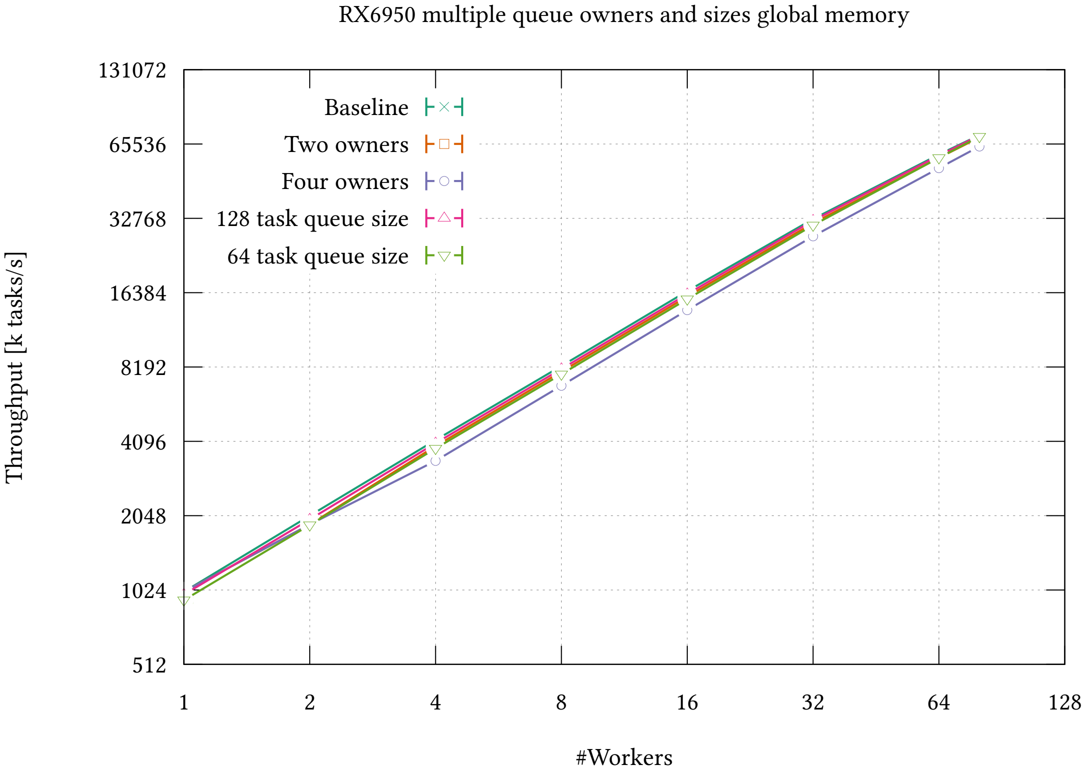

Figure 11 : Compares performance of workers sharing ownership of public queues and different public queue sizes with the queues stored in global memory using the RX6950 loading initial tasks set from global memory. The baseline has public queues with a size of 256 and no ownership sharing. All configurations exhibit similar performance, but reducing the queue size offers better performance than increasing the number of owners per queue. The performance differences stay consistent as the worker count increases.

6.7. Multiple Queue Owners

MGWS supports sharing ownership of public queues among multiple workers, which can reduce the space needed by MGWS at the cost of increased contention on the public queues. An alternative to sharing public queue ownership is to reduce the public queue size. In this section we evaluate the overhead introduced by sharing a public queue among two and four workers using the MEMSET workload against a baseline where each worker has its own queue as well as configurations where the public queue has a size of 256, 128, and 64 tasks, corresponding to the full size, half size, and quarter size. The results are evaluated with the orchestration data stored in global memory, remote global memory, and host memory.

Global memory: Figure 11 shows the global memory measurements. The baseline, meaning each worker has its own queue with a size of 256 tasks, has the highest tasks per second. The second-fastest configuration is using queues with a size of 128 tasks, which is up to 3.56% slower; then comes two-way ownership sharing, which is at most 8.54% slower, followed by using queues with a size of 64, which is up to 9.07% slower, and finally four-way ownership sharing, which is up to 21.36% slower. The measurements show that both reducing the size of public queues and increasing the owners per public queue reduce the number of tasks per second that can be computed. Notably, increasing the number of owners has a greater overhead

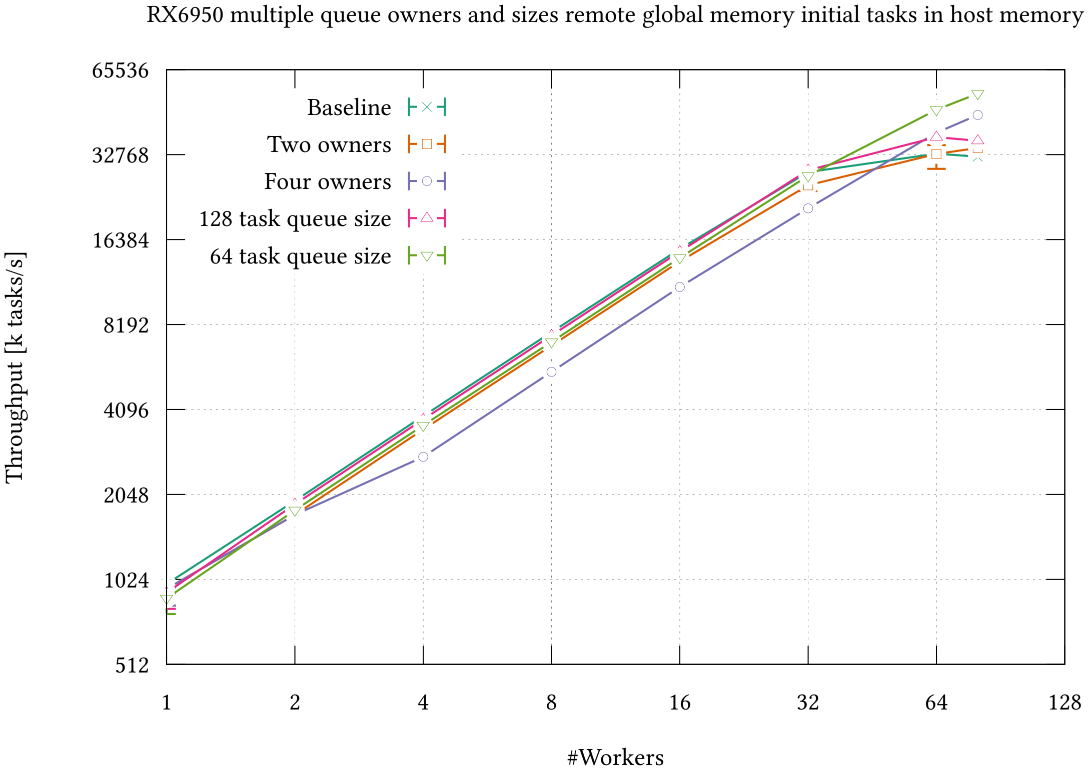

Figure 12 : Compares performance of workers sharing ownership of public queues and different public queue sizes on the RX6950 using the global memory of the RX6600 via peer-to-peer DMA to store the queues. In this setup the initial task set resides in host memory. The baseline has public queues with a size of 256 and no ownership sharing. At low worker counts, all configurations except four-way ownership sharing are close, and reducing the queue size offers better performance than increasing the number of owners per queue. At high worker counts this changes, and queues with a size of 64 tasks and four-way sharing become the best configurations.

than reducing public queue size. We expected that reducing the public queue size or increasing the number of owners per queue would reduce performance because the public queues become empty more quickly, and thus steal attempts and loads from the initial task set grow more frequent. Additionally, increasing the owners per queue causes extra overhead because only one owner can access its public queue at a time.

Remote global memory: The remote global memory evaluation is divided into two figures: Figure 12, which shows tasks loaded from host memory, and Figure 13 which shows tasks loaded from global memory. We present these separately because the relationship between the queue ownership sharing and queue sizes differs depending on the location of the initial task set.

In contrast, the local global memory and host memory configurations do not exhibit such variations based on the location of the initial task set. We, therefore, omit these measurements and focus exclusively on the tasks initially residing in global memory, as this configuration yields better performance.

Starting with the initial task set in host memory, up to 32 workers, the measurements are similar to the local global memory configuration. However, the performance gap between the configurations is more significant, and the 64-task queue size outperforms the two-way owner-

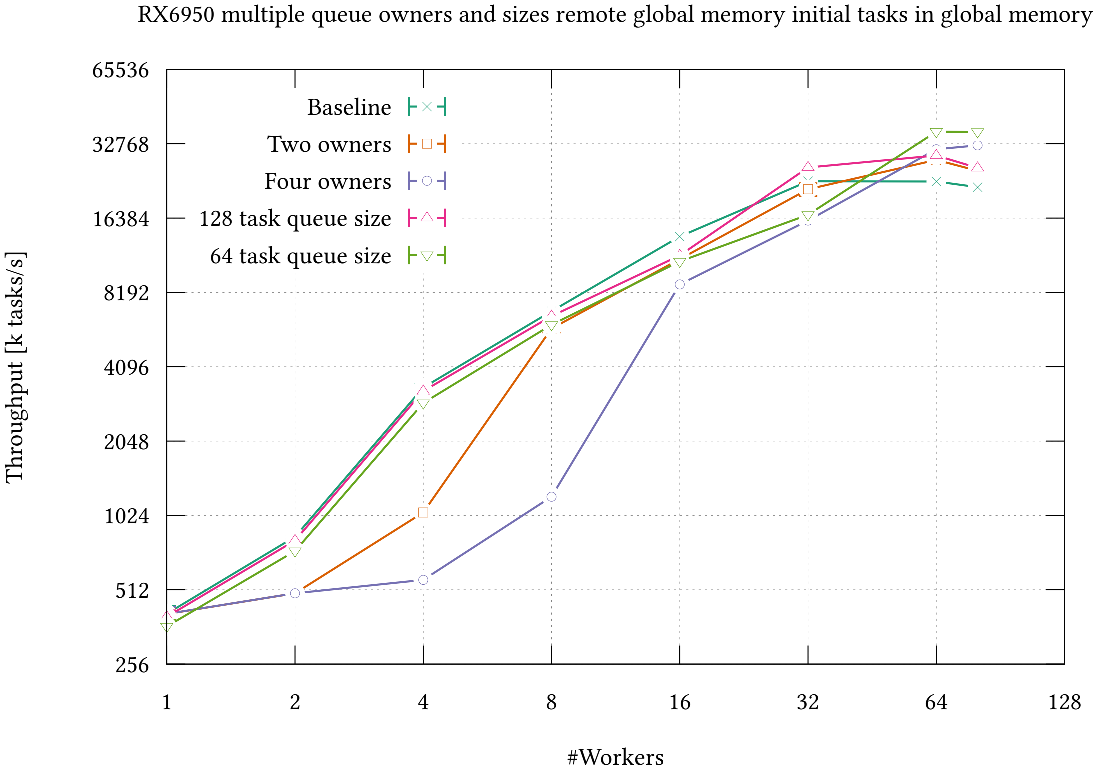

Figure 13 : Compares performance of workers sharing ownership of public queues and different public queue sizes on the RX6950 using the global memory of the RX6600 via peer-to-peer DMA to store the queues. In this setup the initial task set resides in the global memory of the RX6600. The baseline has public queues with a size of 256 and no ownership sharing. At low worker counts, four-way and two-way ownership sharing are significantly worse than the other configurations. At high worker counts, however, this changes, and queues with a size of 64 tasks and four-way sharing become the best configurations.

ship sharing, which is not the case in the global memory setting. As before, the baseline remains the fastest configuration in this interval, followed by the reduced queue size configurations with sizes of 128 and 64 tasks, which are up to 2.72% and 8.68% slower than the baseline, respectively. In the same interval, the ownership-sharing configurations perform the worst, with the two-way sharing being up to 12.29% slower and the four-way being up to 39.87% slower than the baseline.

However, beyond 32 workers, this behavior changes. At high worker counts, all configurations except those that most aggressively reduce the effective per-worker queue size, namely the 64-task queues and the four-way ownership sharing, exhibit a decline in performance when scaling from 64 to 80 workers. Thus resulting in a 29.06% and 40.49% performance lead for the four-way ownership sharing and 64-task queues, respectively, compared to the baseline. Based on the baseline experiencing this slowdown as well, we initially expected all configurations to show a similar performance decline. However, this is not the case. We hypothesize that the slowdown is caused by PCIe bandwidth limitations at high worker counts. If this hypothesis holds, the small task queues and four-way ownership sharing configurations may act as a form of damping mechanism: with four workers sharing a queue, only one worker can interact with the public queue at a time, forcing the others to wait and thereby reducing concurrent PCIe

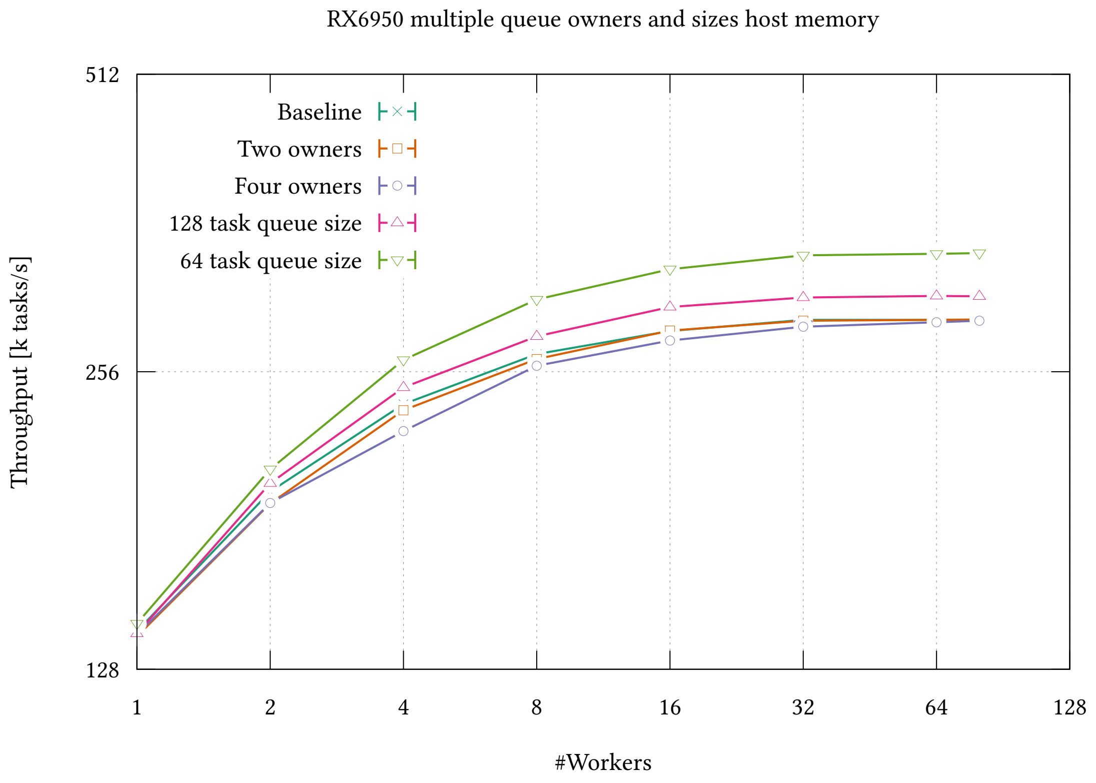

Figure 14 : Compares performance of workers sharing ownership of public queues and different public queue sizes with the queues stored in host memory using the RX6950, loading the initial task set from global memory. The baseline has public queues with a size of 256 and no ownership sharing. Reducing the queue size increases performance compared to the baseline. Smaller queue sizes outperform the baseline.

traffic. Similarly, limiting queues to at most 64 tasks reduces the number of tasks that can be acquired or stolen concurrently, which may help to distribute the PCIe bandwidth more evenly across workers, thereby improving overall performance.

Next we discuss the measurements obtained when tasks are initially located in remote global memory. We divide these measurements into three phases. The first phase extends up to eight workers and exhibits a severe performance degradation, with two owners per queue being up to 3.21 times and four owners per queue up to 6.01 times slower than the baseline. In general, when using remote global memory with tasks initially residing on the remote device, we observe a significant performance difference compared to loading initial tasks from host memory, particularly for configurations with one to two workers. The difference diminishes at four workers; however, using multiple workers per queue appears to exacerbate the resulting performance dip.

The second phase encompasses 16 to 32 workers. It sees the ownership-sharing configurations closing the gap to the other configurations, and two-way ownership sharing even overtaking task queues with 64 elements. Finally, the third phase consists of the remaining measurements, 64 to 80 workers. This phase is similar to the configuration where the tasks are initially stored in host memory as the 64-element task queues and four-way ownership sharing become the best-performing configurations. The baseline becomes the worst configuration, eventually being 47.69% slower than the four-way ownership sharing and 67.55% slower than the

64-task public queues. As discussed for the case where tasks are loading from host memory, we believe the performance advantages at high worker counts are likely caused by PCIe bandwidth limitations. However, additional experiments are required to verify this hypothesis, for example, by reducing the number of available PCIe lanes.

Host memory: Figure 14 depicts the same configurations but uses host memory to store the orchestration data. We initially expected these results to be similar to the global memory setup. However, decreasing the size of the public queue actually increases the tasks per second by up to 16.81% compared to the baseline instead of decreasing it. The second-fastest configuration has queues of size 128 and is up to 5.75% faster compared to the baseline.

In this scenario, public queues are located in host memory, and when a worker retrieves new tasks from the initial task set, it fills the public queue completely and the local queue halfway. Thus, tasks that are not loaded into the local queue are first copied from the task set in global memory to the public queues in host memory and finally acquired and loaded into the local queue in shared memory. Decreasing the size of the public queue therefore also decreases the number of tasks that must be copied into host memory and increases the relative portion of tasks that are directly loaded into the local queue. These factors may explain the performance benefits of having smaller public queues, at least for the MEMSET workload.

However, the effect does not translate to the ownership sharing; two-way ownership sharing is between 0.01% at 80 workers and 2.82% at two workers, slower than the baseline. Four-way ownership sharing of queues is between 6.48% at four workers and 0.22% at 80 workers, worse than the baseline, while having multiple queue owners also reduces the number of tasks each worker must copy between the task set and public queue; the additional communication overhead seems to outweigh this benefit.

Discussion: We set out to answer which of the configurations may be the best to reduce MGWS's memory footprint while still keeping a relatively high performance. However, instead we observe that, depending on the memory configuration and number of workers, reducing queue sizes or increasing the number of per-queue owners may even increase performance. The observed behavior is surprising and indicates that, independently of whether it is better to reduce queue size or increase the number of owners, certain configurations may perform better or worse depending on the inter-worker communication scheme.

6.8. Single-GPU Evaluation

The following section presents the single-GPU evaluation of MGWS. Using a single GPU allows us to evaluate MGWS while facilitating inter-worker communication over global and remote global memory, which, because of hardware limitations, is not possible in the multi-GPU setup. We first compare the performance of the RX6600 and RX6950 to determine whether per-worker performance is similar. Thereafter we assess the performance of the MEMSET workload on the RX6950 and compare MGWS with a static task distribution. Finally, we discuss the performance of the irregular CONTAINS workload and again compare MGWS to a static task distribution.

6.8.1. Comparison Between RRX6950 and RX6600

This section compares the performance of our two GPUs shown in Figure 15 to determine whether they exhibit similar per-worker performance. Both GPUs share the same RDNA 2.0 architecture with similar clock speeds. The most significant difference, apart from the RX6950 having more compute units, is the VRAM. Both GPUs use GDDR6 memory; the RX6950 contains

16 GB with a bandwidth of 576 GB/s [14], and the RX6600 contains 8 GB with a bandwidth of 224 GB/s [15]. Nevertheless, we expect per-worker performance to be nearly identical between the GPUs. Our measurements confirm this assumption, as performance across the GPUs is nearly identical. Under the regular workload, the largest observed difference is 5.77% between the global memory setups with the tree-based termination detection. We observe similar behavior under the irregular workload with 200 repetitions. In this case, the largest performance difference occurs at a single worker, where the RX6950 is 13.25% faster; however, the overall difference is not significant given that the measurements frequently exhibit substantial outliers. Thus, the additional memory bandwidth does not translate to improved performance in our use case. Based on this observation, we limit our further single-GPU evaluation to the RX6950, which can support a larger number of concurrent workers.

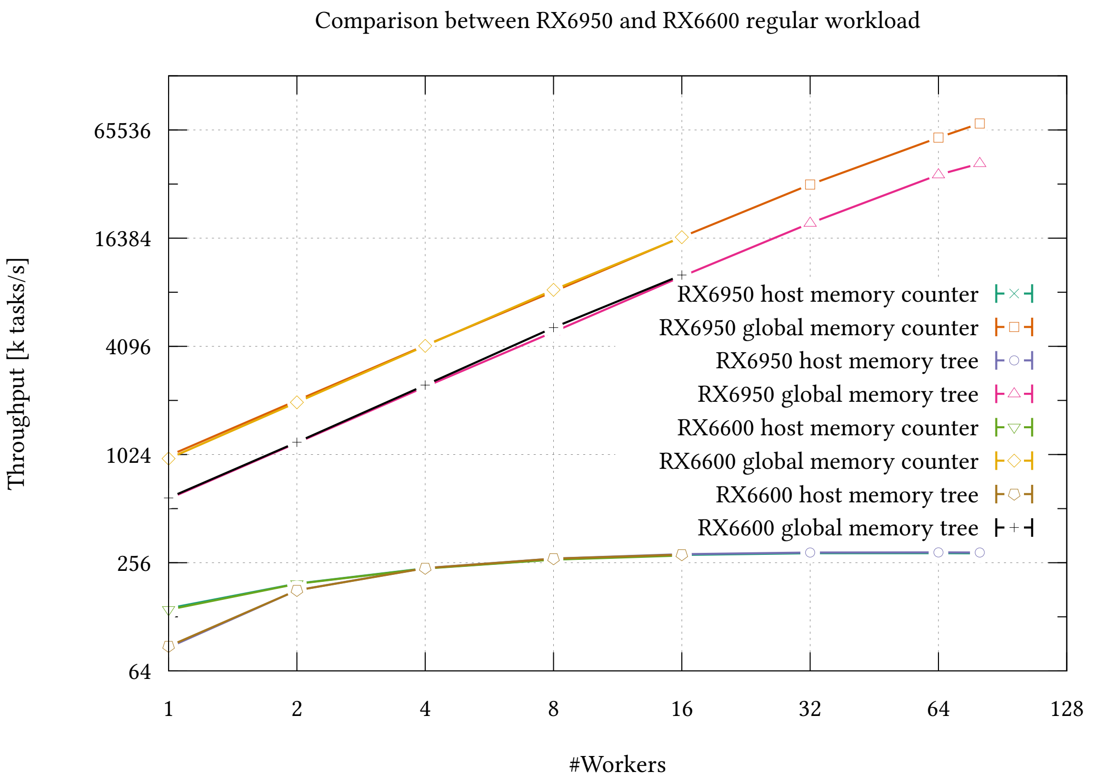

Figure 15 : A performance comparison between the RX6950 and RX6600 under the regular workload. Four configurations are evaluated on each device, combining orchestration data in host or global memory with the tree-based and counter-based termination detection mechanisms. The two GPUs exhibit nearly identical performance across all configurations; consequently, the plots for both GPUs overlap. Additionally, the configurations using host memory also exhibit similar performance for high worker counts, causing the corresponding plots to overlap as well.

6.8.2. RX6950 Regular Workload

This section assesses the performance implications of having orchestration data, such as public queues and the initial task set, located in global memory, host memory, or in the global memory of the other GPU. Storing the orchestration data on another GPU provides an estimate of

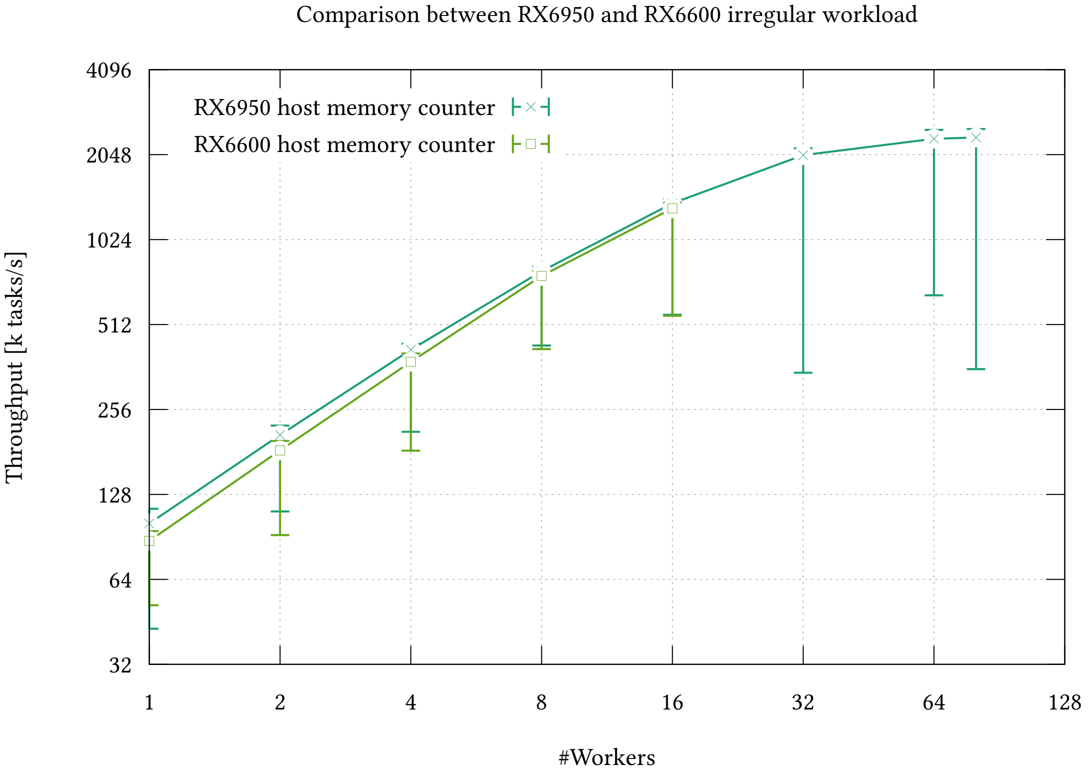

Figure 16 : A performance comparison between the RX6950 and RX6600 under the irregular workload. The two GPUs exhibit nearly identical performance.

what performance is to be expected from the peer-to-peer DMA-based communication scheme. However, we would expect the real peer-to-peer DMA communication scheme to exhibit better performance because in this experiment all public queues are located on the other GPU, whereas normally public queues are stored on the same GPU as the workers they belong to. Thus, in this setup, release and acquire operations need to access the other GPU, which is normally not required, and all steals are similar to stealing tasks from a worker located on another GPU.

We use the MEMSET workload since we are interested in the upper limits and scaling of MGWS. We also compare MGWS to loading and executing the tasks directly. The measurements can be seen in Figure 17. For MGWS, six configurations are examined, combining orchestration data in host, global memory, or remote global memory with initial tasks in host or global memory. In the remote global memory configuration, the initial task set is also located in remote global memory instead of local global memory. We first examine MGWS and then compare it to executing the tasks directly.

Our measurements show that the location of public queues is more important than the location of the initial task set. However, having the task set in global memory yields an improvement of up to 25.91% compared to having the initial task set located in host memory when using global memory for orchestration. These results are to be expected because the tasks do not have to be loaded from host memory when initially filling the queues. Surprisingly, when using remote global memory, having the tasks in host memory is between 9.13% and 58.47% faster than having them also in remote global memory. We suspect that copying data from host memory to remote global memory may be faster than copying data inside remote memory, but we did not verify this hypothesis. However, this would explain why having the initial task set in host memory

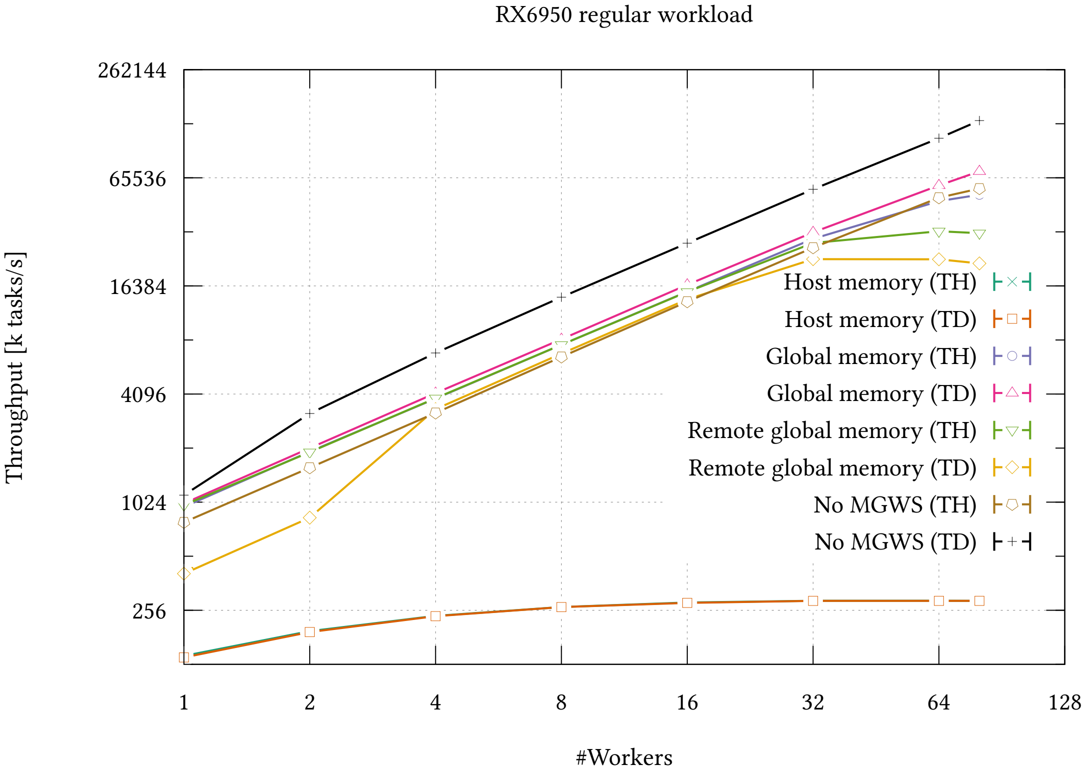

Figure 17 : The number of MEMSET tasks the RX6950 can complete with an increasing number of workers. The figure compares multiple configurations: *Host memory*, *Global memory*, and *Remote global memory* indicate where the orchestration data is stored, while *Task Host (TH)* and *Task Device (TD)* denote the location of the initial task set, which may reside in host memory or global memory. In the remote global memory configuration, TD specifically refers to remote global memory rather than local global memory. We also measure the number of tasks that can be executed per second when loading and executing tasks directly without MGWS. Storing the orchestration data in host memory has a severe impact on the number of completed tasks per second, while the location of the initial task set has a smaller impact but is still noticeable. The host memory configurations exhibit similar performance, causing the *Host memory (TH)* and *Host memory (TD)* plots to completely overlap.

yields better performance. When the orchestration data resides in host memory, the location of the initial task set is insignificant.

Having the orchestration data and thus the public queues in global memory is up to 246 times faster than having the queues reside in host memory, which is significant, and we believe it is caused by host memory bandwidth limitations. Our reasoning, along with a discussion of other potential causes for this performance gap, is provided in Section 6.10.

The remote global memory performance is between 2.11% at one worker and 120.97% at 80 workers slower than using local global memory but up to 114 times faster than host memory. These measurements indicate that inter-GPU communication via peer-to-peer DMA will likely be similar to global memory performance and far exceed the host memory-based approach.

When the public queues are located in global memory, we observe close to a linear speedup until 32 workers and until 16 workers when using remote global memory, whereas when using host memory, we observe a logarithmic speedup with little to no gains after 16 workers.

The slower speedup after 32 when using global memory with tasks in host memory may also be related to host memory bandwidth limitations, because the global memory setup, which loads tasks from host memory, is more affected than the configuration that does not need to load tasks from host memory. Alternatively, the PCIe bandwidth could be a limiting factor because we observe similar slowdowns using the remote global memory configuration, which does not use host memory.

Executing tasks from host memory without MGWS also has less speedup after 64 workers, which supports our hypothesis that a memory or PCIe bottleneck may be to blame. A third possibility is that the VRAM bandwidth becomes the limiting factor as the number of workers increases, since the global memory configuration, which avoids loading tasks from host memory, also has less speedup after 32 workers. We did not have the opportunity to confirm this as part of our thesis; however, future work could investigate this by underclocking the VRAM or reducing the number of PCIe lanes.

Next, we compare MGWS to a static task assignment. When executing the MEMSET tasks without MGWS, we assign each block an equal portion of the tasks. Every block loads and executes the tasks one at a time, either from host memory or global memory. As discussed before, having the MGWS orchestration data in host memory is severely slower compared to storing this data in global memory; we therefore focus on the comparison between the orchestration data in global memory and directly loading tasks since they align more closely.

Executing tasks without MGWS from host memory with up to 32 workers is between 12.03% and 22.94% slower than using MGWS; thereafter, directly executing the tasks is up to 7.63% faster. MGWS outperforming direct task execution in this scenario is probably because the tasks are loaded individually rather than in bulk, as is done in MGWS. However, at high worker counts, the overheads caused by the dynamic load balancing outweigh this advantage. Notably, directly executing tasks from host memory is consistently 17.05% to 28.93% slower compared to using MGWS with the tasks in global memory.

Both comparisons show that even for regular tasks, MGWS can outperform a naive task loading and execution implementation. However, MGWS and directly executing tasks are the fastest when the tasks are already in global memory; here, directly executing the task is up to 47.84% faster compared to MGWS. We expected the direct execution to be faster than MGWS, since the focus of MGWS lies on irregular tasks and the dynamic load balancing causes overhead. However, further low-level optimizations, such as reducing intra-block synchronization, may help to narrow the performance gap. Furthermore, using host memory for inter-worker communication incurs substantial overhead compared to using global or remote global memory. The remote global memory performance indicates that using peer-to-peer DMA for inter-GPU communications likely performs near global memory performance and does not encounter the same limitations observed when using host memory.

6.8.3. RX6950 Irregular Workload

In this section, we focus on the irregular CONTAINS workload. As seen under the regular workload, we compare MGWS with orchestration data in host memory, global memory, and remote global memory to a static task assignment evenly distributing tasks among workers without MGWS. Compared to the MEMSET workload, under the CONTAINS workload, the task execution time dominates overall runtime. Additionally, the CONTAINS workload is memory bound; thus,

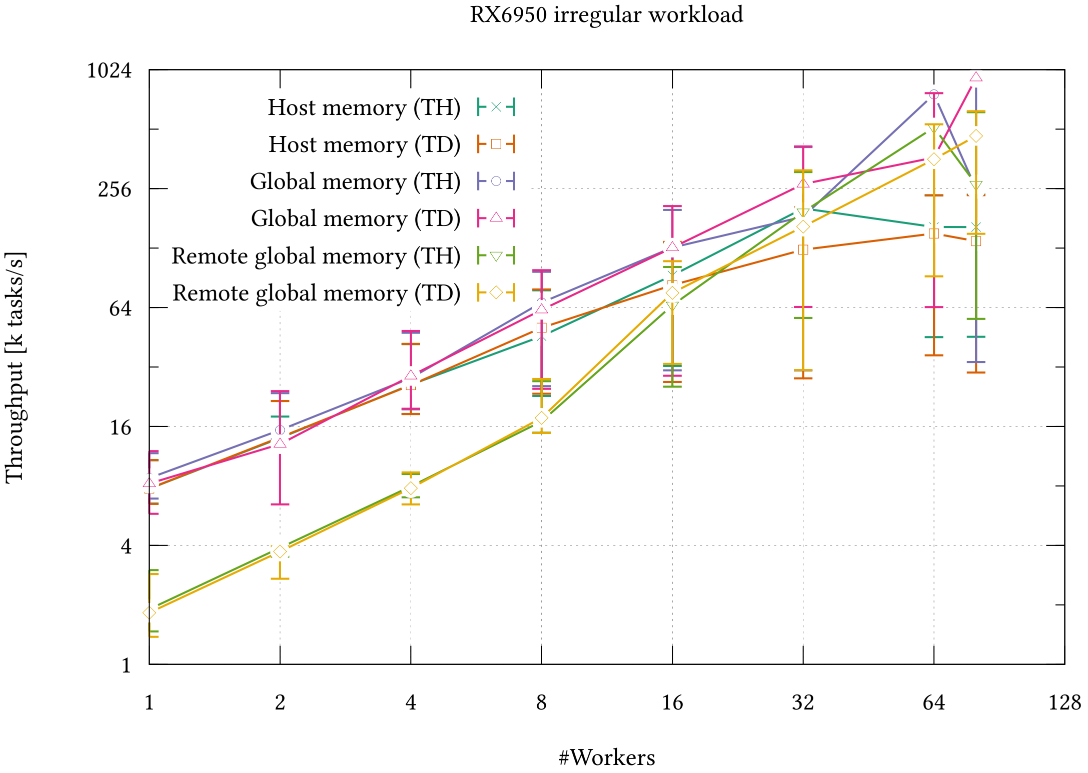

Figure 18 : A performance comparison of the irregular CONTAINS workload on the RX6950 between MGWS configurations with orchestration data residing in *Host memory*, *Global memory*, and *Remote global memory*. *Task Host (TH)* and *Task Device (TD)* denote the location of the initial task set, which may reside in host memory or global memory. In the remote global memory configuration, TD specifically refers to remote global memory rather than local global memory. All configurations exhibit high variance between runs.

variations in bandwidth latency or cache effects affect this workload more than the MEMSET workload.

The execution times vary significantly more than seen when executing the regular MEMSET workload. More variance is to be expected because, depending on the set of tasks a given worker executes, the runtime can also vary since workers cannot steal from other local queues. Thus, smaller local queues may help to reduce the variance. However, we did not expect variation to such an extent. Even the static task assignment without MGWS exhibits large differences between runs, which is surprising given that each worker receives the same set of articles and the same search term every run. Upon closer investigation, we observed that several runs take between 10 and 20 times longer than the average. Such large performance variations are unlikely to be caused solely by memory and cache effects. Thus, further experiments are required to locate the cause. Since these outliers also occur without MGWS, this indicates that they are caused by the tasks themselves rather than by MGWS. Nevertheless, a higher-quality random number generator for target selection could reduce variance when using MGWS. We use hipRAND to generate random numbers, but we have not evaluated their quality.

The measurements are divided into two figures to increase legibility and to highlight the comparison of MGWS to direct task execution. Figure 18 compares the global memory, remote global memory, and host memory MGWS configurations with the initial task set located either

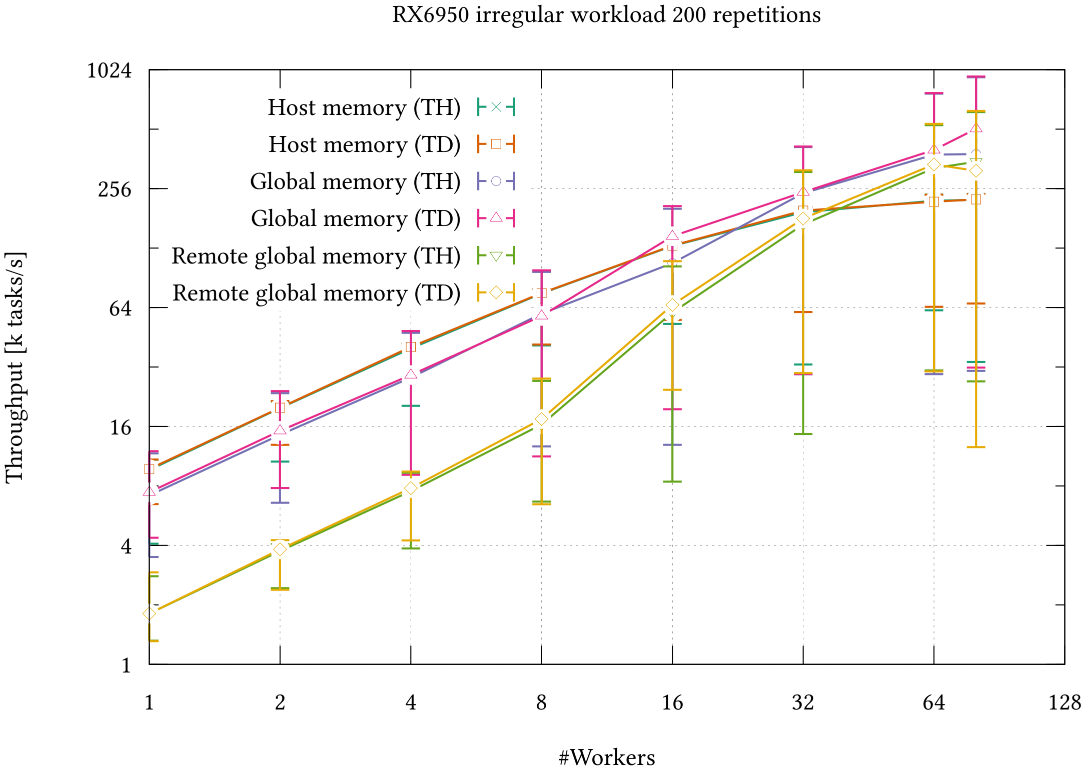

Figure 19 : A performance comparison of the irregular CONTAINS workload on the RX6950 between MGWS configurations with orchestration data residing in *Host memory*, *Global memory*, and *Remote global memory*. *Task Host (TH)* and *Task Device (TD)* denote the location of the initial task set, which may reside in host memory or global memory. In the remote global memory configuration, TD specifically refers to remote global memory rather than local global memory. Each experiment is repeated 200 times to reduce the impact of outliers and obtain representative average performance. The measurements show that at high worker counts, using global memory is the fastest configuration, followed by remote global memory and host memory. The host memory configurations exhibit nearly identical performance, causing the plots to overlap.

in host memory, global memory, or remote global memory. Figure 20 focuses on comparing the MGWS configuration with orchestration data in global memory, which is the best-performing MGWS configuration, against direct task execution using a static task assignment. The static task assignment gives each worker an equal number of tasks. Both configurations are evaluated in two variants: one where the initial task set resides in host memory and one where it resides in global memory.

To reduce the influence of outliers, we also provide both figures with 200 instead of 10 repetitions. The comparison between MGWS configurations is shown in Figure 19, while the comparison between MGWS and direct execution is depicted in Figure 21.

We first take a brief look at the trends observed when comparing the MGWS configurations using 10 repetitions and then investigate whether these trends are also visible when using 200 repetitions. Up to eight workers, the host, and global memory configurations are close, while the remote global memory configurations are between 4.24 times and 2.59 times slower. From 16

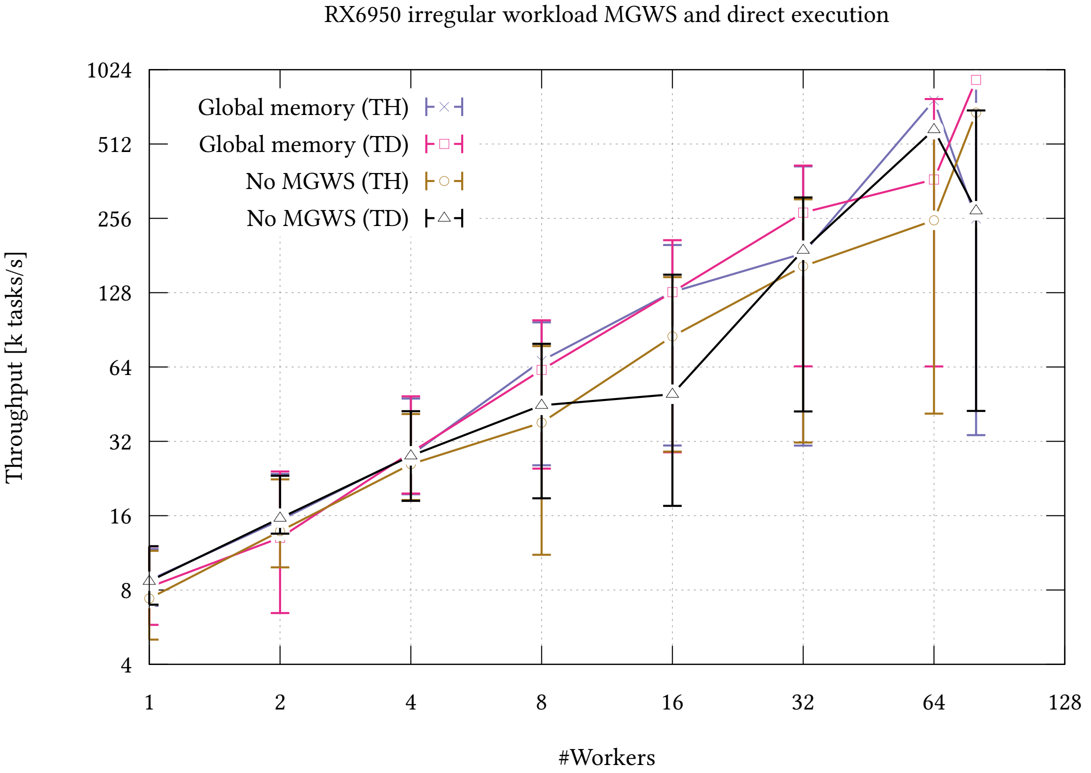

Figure 20 : A performance comparison of the irregular CONTAINS workload on the RX6950 between MGWS with the orchestration data residing in global memory against direct task execution without MGWS. *Task Host (TH)* and *Task Device (TD)* denote the location of the initial task set, which may reside in host memory or global memory. All configurations exhibit high variance between runs. For most measurements, MGWS outperforms direct task execution.

workers onwards, this trend reverses: at high worker counts, the global memory configurations are the fastest, followed by the remote global memory, with host memory being the slowest.

We now analyze the same configurations using 200 repetitions. Due to the high fluctuations, we focus on comparing the average performance. Up to eight workers, the host configurations achieve the highest performance. The global memory configurations perform similarly, both approximately 30% slower than host memory. The remote global memory configurations are between 3.76 times and 2.78 times slower than the others. It is surprising that the host memory configurations outperform the global memory configurations, likely because they experience fewer outliers, though the cause for the outliers remains unclear.

Similar to the measurements obtained using 10 repetitions, the host memory configurations lose performance relative to the others from 16 workers onwards. At 80 workers, the global memory configurations achieve the highest performance, followed by the remote global memory configurations, which are 9.34% slower. The host memory configurations perform substantially worse, exhibiting slowdowns of 69.65% compared to global memory. We did not expect the remote global memory configuration to be so close to the other configurations because in this configuration the corpus is stored on the remote GPU rather than in local global memory. Thus, in this configuration, tasks load the articles from the remote GPU instead. The results suggest that, when peer-to-peer DMA is available, data migration between GPUs may not be as critical as initially anticipated.

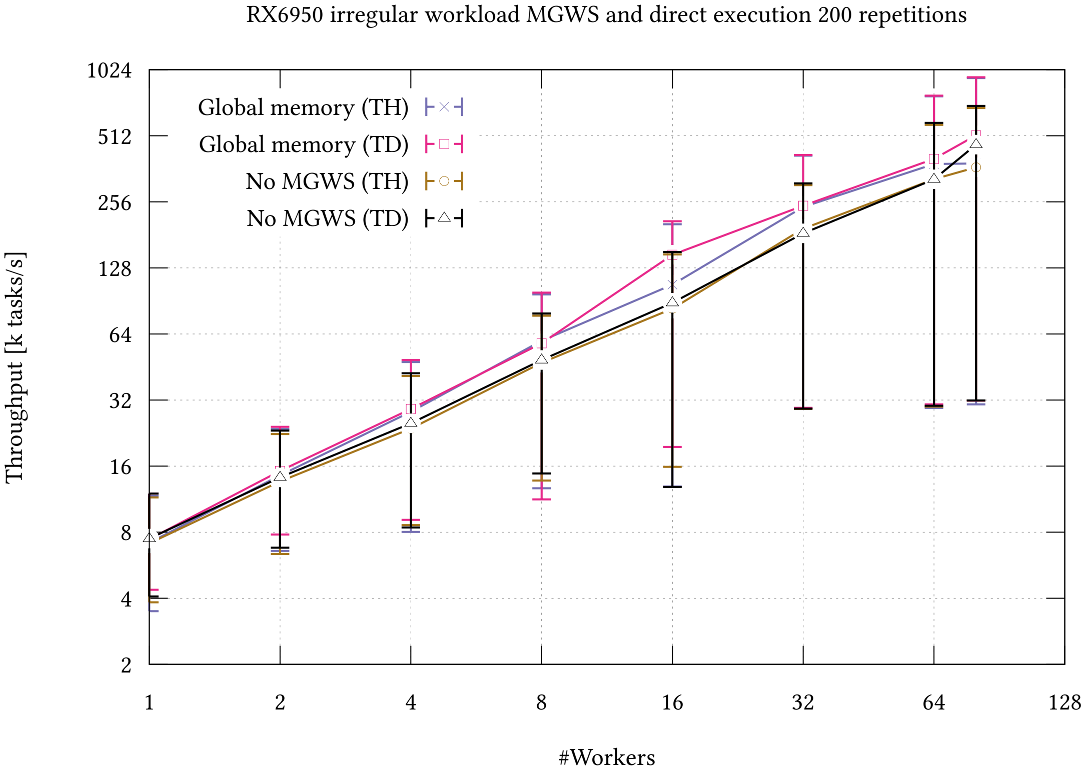

Figure 21 : A performance comparison of the irregular CONTAINS workload on the RX6950 between MGWS with the orchestration data residing in global memory against direct task execution without MGWS. *Task Host (TH)* and *Task Device (TD)* denote the location of the initial task set, which may reside in host memory or global memory. All configurations exhibit high variance between runs. Each experiment is repeated 200 times to reduce the impact of outliers and obtain representative average performance. Using MGWS on average offers better performance than using a static task assignment.

Overall, the performance of the different configurations is more similar, with the host memory configuration, in particular, being significantly closer to the other configurations than observed in the MEMSET workload. The smaller differences are likely due to the MEMSET workload being designed to highlight overheads caused by MGWS, but using the CONTAINS workload, the tasks dominate the total runtime. Thus, the additional overheads caused by MGWS using host memory become less significant.

From our measurements, we draw three conclusions. First, the overhead caused by using host memory to store the orchestration data becomes less significant when executing irregular tasks, which dominate overall runtime. Second, based on the remote global memory configuration, we assume that the proper peer-to-peer DMA-based inter-GPU communication scheme will perform close to the local global memory setup. Third, inter-GPU data migration may be less influential when peer-to-peer DMA is available to access data residing on the other GPU.

Next we discuss the difference between using MGWS and direct task execution. As before, we first highlight the trends observed using 10 repetitions and then analyze how these translate to the 200 repetitions. Up to four workers, all configurations exhibit similar performance. Beyond four workers, at least one of the MGWS configurations always outperforms the static task assignment.

Using 200 repetitions, we observe similar behavior. As MGWS, configurations are consistently between 6.56% and 39.57% faster than direct task execution, except with a single worker, where direct execution is 0.64% faster. Specifically, at 80, the difference is 9.3%. The results indicate that MGWS is capable of load-balancing tasks across workers, which causes it to outperform a static task distribution.

We did not quantify the degree of irregularity of the CONTAINS workload. Consequently, in future evaluations with other workloads, the performance difference between MGWS and direct task execution may widen as the workload irregularity increases. As indicated by the error bars, most measurements include extreme outliers towards worse performance, with the averages being noticeably closer to the upper performance limits. Although we are using 200 repetitions, the outliers still impact average performance significantly. We are unsure what causes these outliers, but it is unlikely that MGWS is responsible since direct execution features similar outliers.

RX6950	RX6600	CPU	Total Worker GPU only	Total Worker GPU and CPU
1	1	1	2	3
2	2	2	4	6
4	4	4	8	12
8	8	8	16	22
16	16	16	32	48
32	16	16	48	64
64	16	16	80	96
80	16	16	96	112

Table 8 : The worker counts used in the multi-device evaluation.

6.9. Multi-Device Evaluation

This evaluation assesses the performance of MGWS in a multi-device environment, evaluating workers running on both GPUs and the CPU. CPU worker participation is limited to using the host memory-based inter-worker communication mechanisms. While the CPU can access global memory through, for example, resizable Base Address Register (BAR) support, using HIP, it is not possible for the CPU to perform atomic operations targeting global memory this way. Consequently, inter-worker communication involving CPU workers relies on host memory. In Chapter 7, we outline two mechanisms that may allow the CPU to participate when using the peer-to-peer DMA communication scheme. The multi-GPU setup can theoretically use peer-to-peer DMA for communications rather than host memory; however, because of hardware limitations, our multi-GPU evaluation is also restricted to host memory-based communications.

We evaluate both the regular and irregular workloads using the same two evaluation setups.

The first setup evaluates multi-GPU work stealing without CPU workers using seven different configurations:

1. Only RX6950
2. Only RX6600
3. RX6950 and RX6600; no restrictions on work stealing
4. RX6950 and RX6600; no work stealing between workers
5. RX6950 and RX6600; only work stealing between workers located on the same GPU
6. RX6950 and RX6600; workers have a 75% chance to steal from workers located on the same GPU
7. No MGWS; each worker is statically assigned the same number of tasks

The second setup includes CPU workers and compares eight different configurations:

1. Only RX6950
2. Only RX6600
3. Only CPU
4. Both GPUs and CPU; no restrictions on work stealing
5. Both GPUs and CPU; no work stealing between workers
6. Both GPUs and CPU; only work stealing between workers located on the same device
7. Both GPUs and CPU; workers have a 75% chance to steal from workers located on the same device
8. No MGWS; each worker is statically assigned the same number of tasks

Both evaluation setups use the same worker counts, shown in Table 8. The worker count on each device is doubled until its limit is reached. Each device executes at most as many workers as it has compute units, or cores in the case of the CPU. Due to an experimental configuration error, the RX6600 was evaluated with at most 16 workers instead of the maximum supported 28. Re-running the experiments was not feasible within the project's timeline; however, this limitation does not invalidate the quantitative conclusions drawn from our results. As a result, once a total of 48 workers is reached when using GPU and CPU workers or 32 when only using GPU workers, only the RX6950 continues to increase its worker count since the other devices reach their limit of 16 workers.

The remaining section first examines the regular MEMSET workload, followed by the irregular CONTAINS workload.

6.9.1. Multi-Device Work Stealing with a Regular Workload

The following evaluation examines the overhead and scaling of MGWS when using multiple GPUs; we also examine the impact of adding CPU workers and compare MGWS to direct task execution. To this end, we utilize the MEMSET workload. Figure 22 shows the multi-GPU performance, while Figure 23 shows the performance when both GPU and CPU workers are used. We first compare the MGWS configurations and then focus on the direct execution.

In the multi-GPU setting, restricting the possible stealing targets has almost no impact. We expected this based on the workload only featuring regular tasks. Additionally, both GPUs have nearly identical per-worker performance, and stealing from a worker located on another GPU incurs no additional overhead when using the host memory-based communication since all

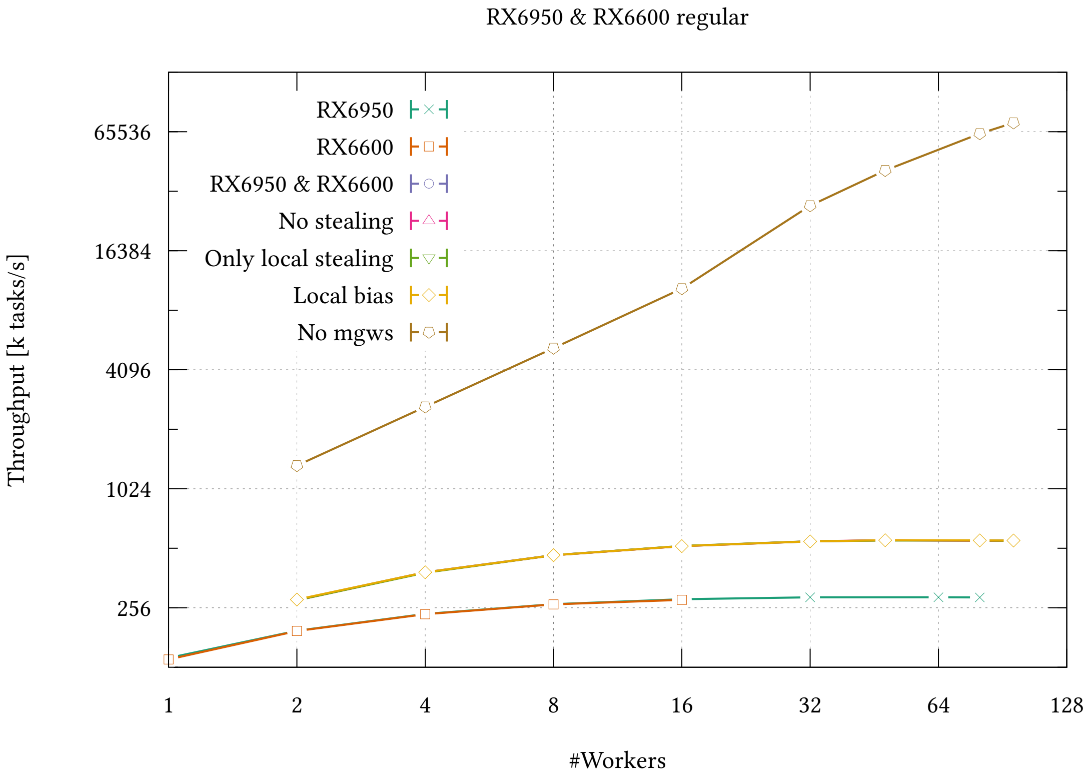

Figure 22 : Shows multi-GPU work stealing performance with the MEMSET workload using the host memory communication scheme and various steal configurations, compared to direct task execution. It also presents the single-GPU performance of both GPUs for reference. All MGWS multi-GPU configurations exhibit similar performance, causing a complete overlap of the plots. Using the host memory-based communication mechanism causes the direct task execution to outperform MGWS significantly.

public queues are located in host memory. All these effects combined explain why restricting stealing has no real impact in this scenario. Another interesting aspect is the comparison between using a single GPU and combining both GPUs. While we would expect a similar per-worker performance compared to the single GPU setup, using both GPUs simultaneously nearly doubles the per-worker performance. The behavior appears to be closely related to the bottleneck limiting host memory-based execution and might be caused by each device having separate host memory bandwidth resources, which is discussed in more detail in Section 6.10.

Direct task execution without MGWS is between 6.78 and 129.8 times faster. We expected this given the single GPU evaluation, where direct task execution performs similarly to using global memory for inter-worker communications, which both are significantly faster than using host memory. However, given that the per-worker performance nearly doubles when both GPUs are utilized, the gap between direct execution and MGWS also nearly halves, from direct execution being up to 246 times faster in the single-GPU setups to at most 129.8 times in the multi-GPU setup.

Attention now turns to the effects observed when adding CPU workers, first comparing the MGWS configurations and then direct task execution. The single-GPU performance is limited by the host memory communications to at most 289 154 tasks per second. On the other hand,

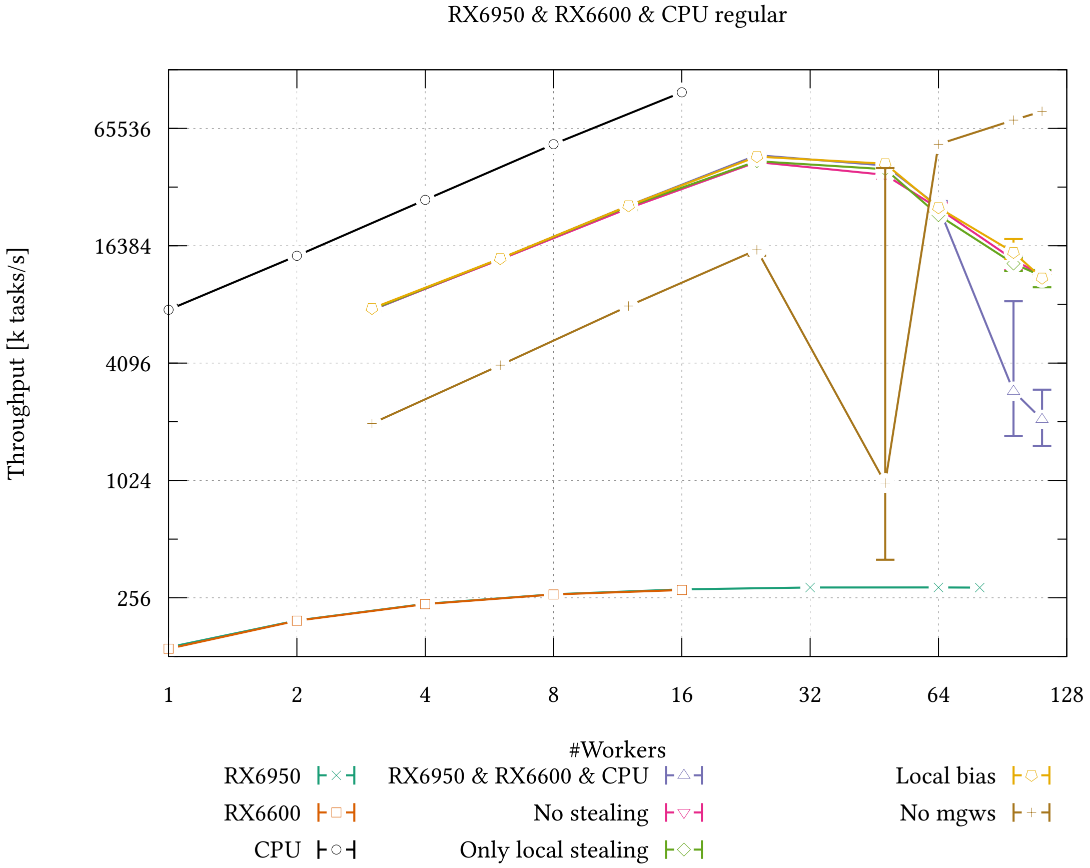

Figure 23 : Shows work stealing performance utilizing GPU and CPU workers with the MEMSET workload using the host memory communication scheme and various steal configurations. MGWS is also compared against direct task execution. It also presents the single-GPU and CPU-only performance for reference. The multi-GPU-CPU configurations perform similarly, leading to a partial overlap of the plots. For worker counts up to 48, MGWS outperforms the direct task execution.

the CPU performance is not restricted to such an extent and achieves linear speedup, reaching up to 98 647 726 tasks per second. The CPU workers being faster than the GPU workers in this scenario is to be expected because the CPU does not rely on the PCIe bus. Moreover, the MEMSET workload is not taking advantage of the parallel nature of GPUs, as it is designed to assess the overhead caused by MGWS itself. Furthermore, the CPU is not affected by the same bottleneck that limits the GPU performance when using host memory for inter-worker communication. Given the performance advantage of the CPU workers, we expect that preventing CPU workers from stealing tasks from GPU workers likely reduces performance. We observe this effect up to a total of 64 workers. In this range, work stealing without restriction is the fastest, only sometimes overtaken by the local bias configuration, whereas no stealing and only local stealing are the slowest. Non-restricted work stealing is up to 10.46% faster than the configuration without any stealing and 9.61% faster than only local stealing.

However, after 64 workers, we observe a reversal; non-restricted work stealing becomes the slowest configuration, even slower than using only a single worker per device, and exhibiting a high variance in tasks per second. We hypothesize this happens because in our setup, the 16-

core CPU is limited to at most 16 workers. Consequently, once the total number of workers exceeds 48, comprising 16 workers on each GPU and 16 on the CPU, the relative portion of CPU workers decreases since the RX6950, with its 80 compute units, can supply up to 80 workers. As a result, after 48 workers, the portion of tasks computed by GPU workers grows since more tasks are loaded into local GPU queues and thus cannot be stolen, leading to a performance peak at 22 to 48 workers. The non-restricted work stealing exacerbates this, leading to more GPU workers stealing tasks from faster CPU workers. Thus reducing the number of tasks that are executed by CPU workers even further and growing closer to the multi-GPU performance without CPU workers. At its lowest, this configuration is 5.53 times slower than the other configurations and 3.64 times slower than using a single worker from each GPU and the CPU. The configurations that restrict stealing decline less, likely because the GPU workers are less likely, or entirely prevented, from stealing tasks from CPU workers. Furthermore, the peak multi-GPU CPU configuration performance never reaches the single CPU performance, being at least 2.1 times slower, which further highlights that tasks executed by the GPU instead of the CPU reduce overall performance.

We want to emphasize that this should only hold true if the tasks heavily prefer the CPU, which is usually not the case when using a GPU work stealing system. Moreover, the host memory-based communication further limits GPU performance. Thus, if this issue can be resolved in the future, for example, by letting GPUs communicate via peer-to-peer DMA, we expect the GPU workers to approach CPU worker performance.

We now compare the direct task execution. Given the measurements without CPU workers, we expected to observe similar behavior in this setup. However, MGWS outperforms direct task execution for up to 48 workers by between 3.89 times and 43.47 times in the multi-GPU environment with CPU participation. Beyond 48 workers, the direct task execution is up to 7.71 times faster, which is still significantly less than observed without CPU workers. We do not know what causes the drastic outliers at 48 workers during direct task execution. However, we ran the experiment multiple times and observed the same behavior every time.

We believe that the performance advantage at small worker counts is caused by the static task assignment giving the CPU workers fewer tasks than they can process using MGWS. The advantage persists even when stealing is prohibited because, using MGWS, the initial task set is not evenly distributed among workers; instead, each worker loads new tasks as soon as its queues become empty. Since CPU workers empty their queues more quickly, they load a larger share of the initial task set. At worker counts above 48, the host memory-based MGWS approach declines as GPU workers begin to significantly outnumber CPU workers and increasingly steal tasks from them. Moreover, at these high worker counts, direct task execution takes the performance lead because, with many GPU workers, the performance gap between the combined GPU workers and the combined CPU workers becomes less pronounced.

In conclusion, MGWS can outperform direct task execution even for regular tasks when the compute performance is unevenly distributed, up to the point where GPU workers begin to heavily outnumber the CPU workers. Nevertheless, even if per-worker performance nearly doubles with both GPUs, the overhead associated with host memory-based inter-GPU communications remains significant, particularly for short tasks.

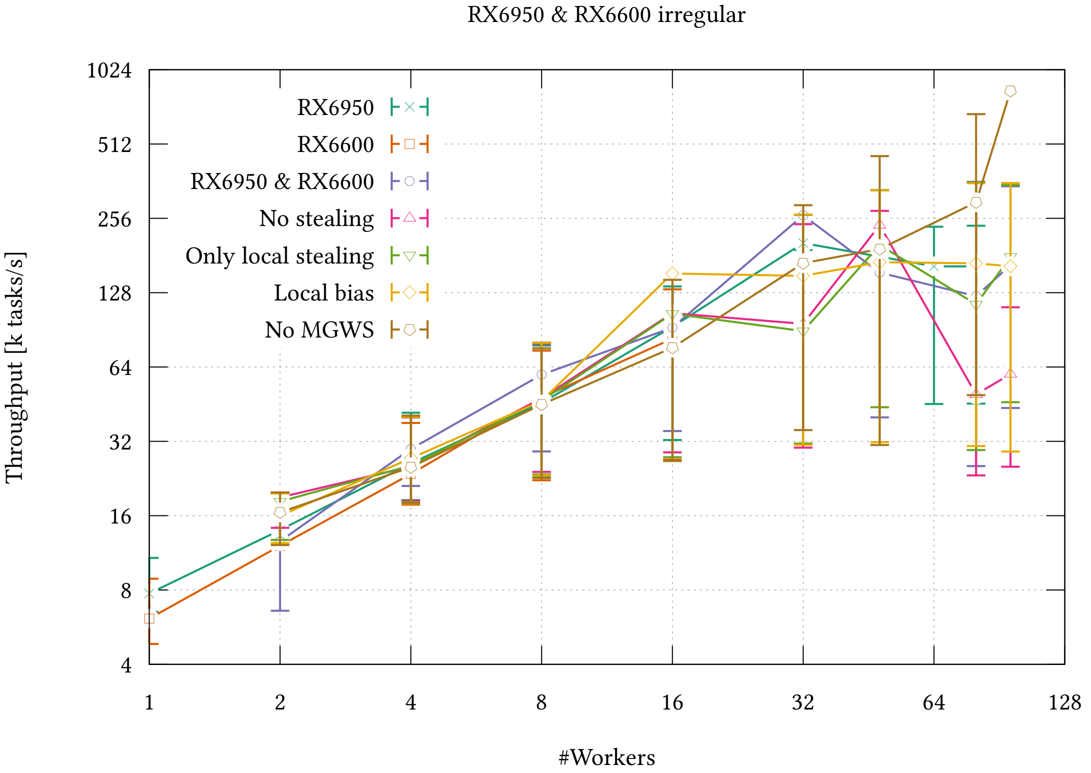

Figure 24 : Shows multi-GPU work stealing performance on the irregular CONTAINS workload using the host memory communication scheme and various steal configurations. It also presents the single-GPU performance of both GPUs for reference. All configurations exhibit high variance between runs.

6.9.2. Multi-Device Work Stealing with an Irregular Workload

Here, we assess the performance of MGWS handling irregular tasks with multiple GPUs and CPU workers participation; to that end, we are executing the CONTAINS workload. We also compare the performance of MGWS to directly executing the tasks via a static task assignment. The experimental setup and worker configurations are identical to those used for the regular tasks; however, the execution behavior differs given the irregularity of the workload and the longer per-task execution time.

We split our measurements up into multi-GPU-only work stealing, shown in Figure 24, and measurements including both GPU and CPU workers, which are shown in Figure 26.

As discussed in the single-GPU evaluation, the CONTAINS workload measurements include significant outliers, which heavily influence the average performance. It is unlikely that these outliers are caused by MGWS since the direct task execution exhibits similar variation between runs. To reduce the influence of outliers, we also provide the multi-GPU measurements using 200 instead of 10 repetitions, shown in Figure 25. We were unable to evaluate the measurements, including CPU workers using 200 repetitions, because instability at such high repetition counts led to incorrect results. However, this is less necessary because the measurements with CPU participation exhibit fewer outliers, particularly for low worker counts. In the remainder of this section, we examine the measurements obtained using only GPU workers and then the effects of adding CPU workers.

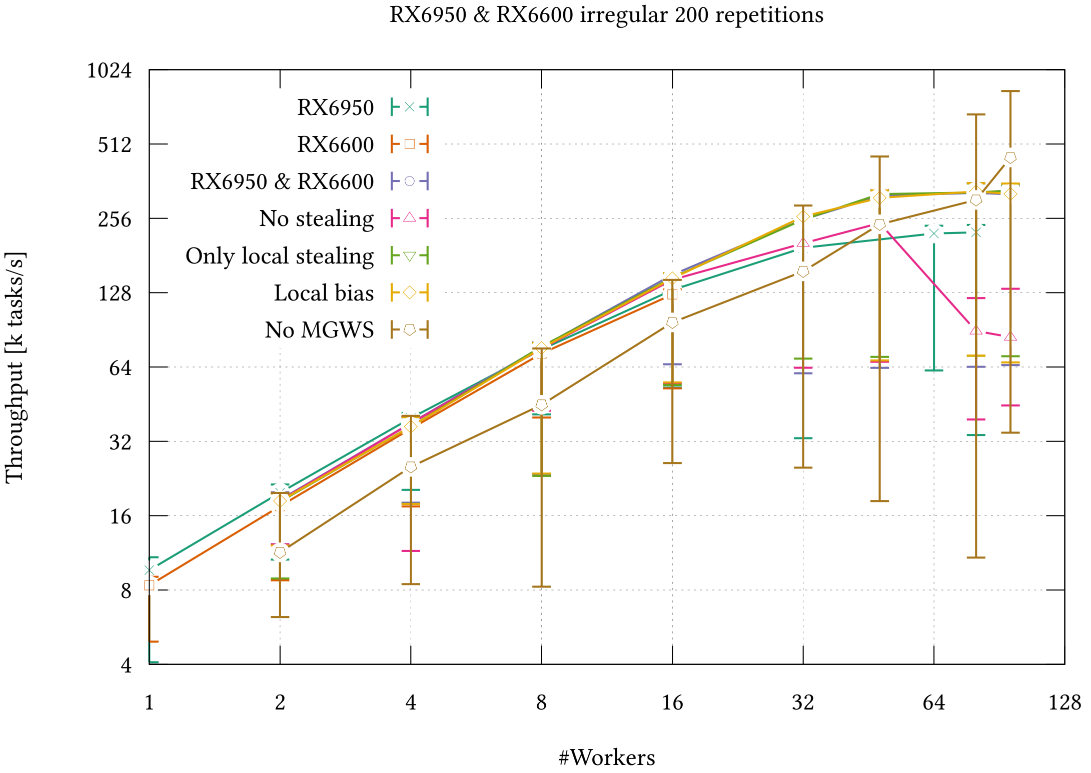

Figure 25 : Shows multi-GPU work stealing performance on the irregular CONTAINS workload using the host memory communication scheme and various steal configurations. It also presents the single-GPU performance of both GPUs for reference. Each experiment is repeated 200 times to reduce the impact of outliers and obtain representative average performance. All configurations except *No MGWS* and *No stealing* exhibit nearly identical performance, causing the plots to overlap.

As part of our multi-GPU analysis, we begin by analyzing the trends indicated using the 10 repetitions and then compare how these trends translate to the 200 repetitions. Compared to the multi-GPU MEMSET measurements we present in Figure 22, we do not observe a significant increase in per-worker performance when using both GPUs instead of just one. Thus, aligning more closely with our initial expectations, which is likely due to the CONTAINS workload being dominated by task execution time; consequently, the MGWS overheads observed in the MEMSET workload become less pronounced.

For up to 16 workers, all configurations exhibit similar average performance. At high worker counts, the differences between the configurations grow, which leads to direct task execution becoming the fastest configuration. Notably, at 96 workers, the direct task execution has no outliers, which likely contributes to its performance lead. This is followed by the three MGWS configurations: one that only allows stealing on the same device, one that has a bias towards workers on the same device, and one that imposes no restrictions on stealing, all of which exhibit similar performance. The configuration that prevents stealing entirely has significantly worse performance.

We proceed to examine the same configurations using 200 repetitions. As before, we observe that most configurations exhibit similar performance up to 16 workers. The only exception is

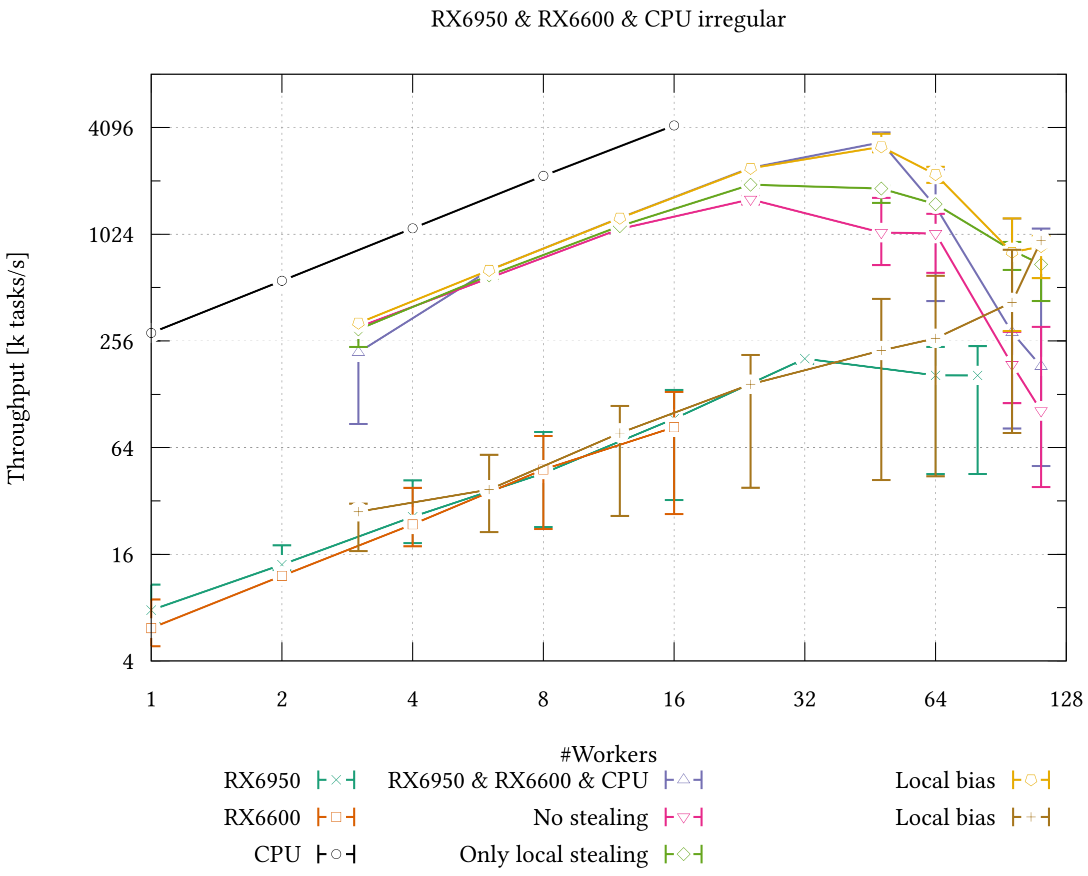

Figure 26 : Shows work stealing performance utilizing GPU and CPU workers on the irregular CONTAINS workload using the host memory communication scheme and various steal configurations. It also presents the single-GPU and CPU-only performance for reference. Most configurations exhibit high variance between runs.

the direct task assignment, which is on average up to 69.76% slower than the others. The poor performance of direct execution at low worker counts mirrors the behavior observed in the single-GPU setup and is likely caused by increased imbalance from the static task assignment, which does not take the varying execution times between tasks into account. At high worker counts, all multi-GPU MGWS configurations except the configuration that prevents all stealing show similar performance. At 96 workers, the direct task execution surpasses MGWS, at which point it is 7.56% faster. The direct task execution outperforming MGWS is likely caused by the host memory-based communication, as no noticeable speedup occurs beyond 48 workers, which is similar to the host memory single-GPU measurements. Noticeably, MGWS approaches the upper limit determined through the MEMSET workload for a single GPU. However, this may be coincidental. In conclusion, MGWS using multiple GPUs can outperform a direct task assignment under the CONTAINS workload for up to 80 GPU workers. Furthermore, all MGWS configurations, except the one that prohibits all stealing, perform nearly identically.

Next we examine the effects of adding CPU workers to the multi-GPU work stealing. Similar to the MEMSET workload, the task favors the CPU; thus, only using the CPU is the fastest. We therefore expect the configuration that restricts stealing the least to perform the best because it allows faster CPU workers to steal tasks from the GPU workers and reduces imbalance caused by the irregular workload. Ignoring the outlier at three workers, this assumption holds up to 48 workers, at which point each GPU and the CPU contribute 16 workers. In this interval, prohibiting stealing entirely is, on average, up to 223.31% slower than unrestricted work stealing, while limiting stealing to the same device is, on average, up to 82.44% slower. Biasing the target selection to favor workers on the same device is on par with the unrestricted stealing, which is likely because the bias is not significant enough to prevent CPU workers from stealing work from GPU workers. Furthermore, CPU workers also load a larger share of the initial tasks set as they empty their queues more quickly. At these lower worker counts, the direct task execution is on average up to 17.30 times slower than MGWS, likely because the CPU executes fewer tasks than it could when allowed to steal work and load a larger share of the initial task set. We also observe significantly less variation between runs when using MGWS.

Notably, all MGWS configurations start to decline after 48 workers, at which point no additional CPU workers are added. The decline is similar to the effect seen when using the MEMSET workload, where after 48 workers, the performance decreases as the portion of tasks executed by GPU workers increases. Configurations that restrict GPU workers from stealing tasks from CPU workers exhibit declines to a lesser extent, except for the configuration that disables all stealing. We hypothesize that the configuration that disables all stealing becomes the worst at high worker counts because, while it prevents slower GPU workers from stealing tasks from faster CPU workers, it also significantly increases the imbalance. The configuration that only allows local stealing also increases imbalance, but this is outweighed by the performance advantage gained by hindering GPU workers from stealing from CPU workers.

At these high worker counts, biasing steal operations towards workers on the same device is the best MGWS configuration. This is followed by only allowing steals to workers located on the same device, which is on average up to 31.98% slower; in the context of CPU workers, *same device* refers to all CPU workers. The worst configurations are non-restricted stealing, which is on average up to 78.95% slower than biased stealing, and disabling stealing entirely, which is on average up to 88.20% slower. Furthermore, the noise of most measurements also increases. The direct task execution does not experience slowdowns after 48 workers and surpasses MGWS at 112 workers, being on average 7.32% faster than MGWS. Without CPU workers, the direct execution is also 7.56% faster than MGWS, which we believe contributes to its performance lead in this setup. Additionally, the direct execution measurement at 112 workers does not exhibit any outlier, a result that remains consistent across multiple retries.

To summarize, MGWS is capable of outperforming a static task assignment for up to 96 workers but experiences slowdown after the GPU workers begin to outnumber the CPU workers significantly. Furthermore, completely prohibiting stealing leads to the worst performance, whereas applying a bias towards workers running on the same device achieves the best performance of all MGWS configurations. Moreover, as noted in the analysis of the multi-GPU CPU MEMSET workload, it is important to emphasize that the evaluation of the CONTAINS workload is affected by the CPU preference of its tasks. Typically, however, tasks in GPU work stealing systems favor the GPU instead. Furthermore, the CONTAINS workload exhibits strong runtime variations, making evaluation and generalization more challenging. Future evaluation with irregular tasks that favor the GPU and are more stable may provide additional insights.

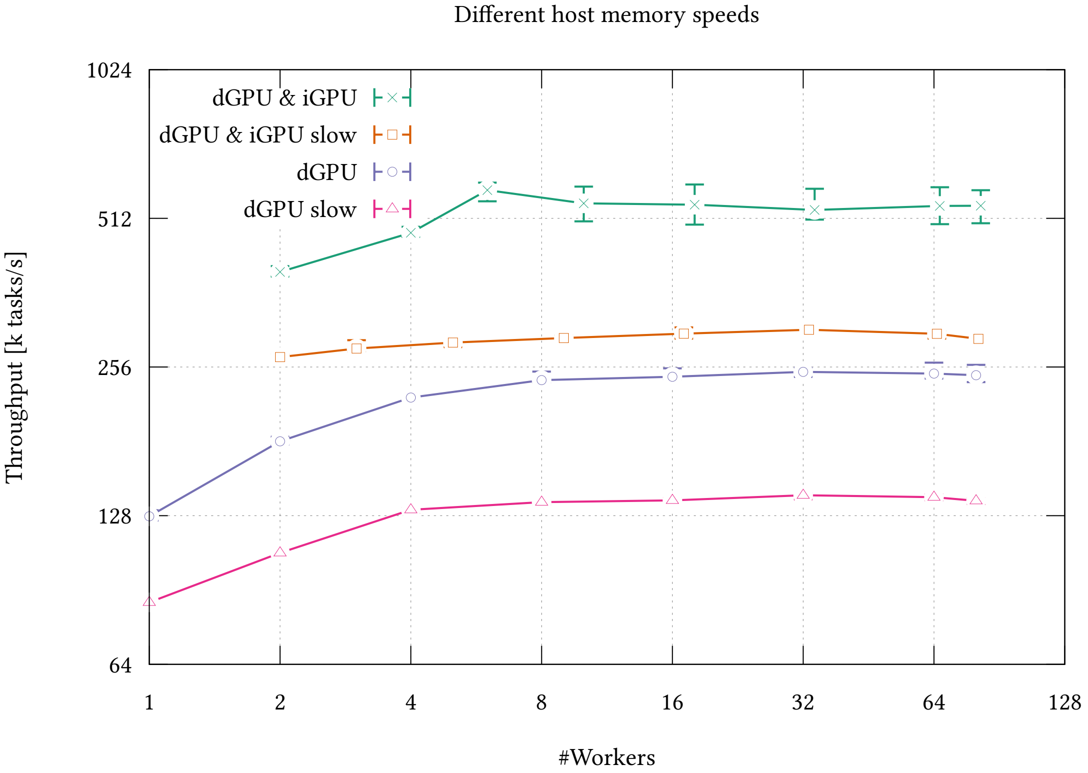

Figure 27 : Shows tasks per second under the MEMSET workload of the desktop. The values marked with slow are run with a reduced DRAM speed of 2400 MT/s compared to the regular speed of 5600 MT/s. The measurements indicate a direct relation between host memory bandwidth and overall performance.

6.10. Understanding the Performance Impact of Host Memory-Based Communication

We identify three possible causes for the drastic performance degradation observed when using host memory for inter-worker communication compared to using global memory. The first is limited host memory bandwidth, the second is PCIe bandwidth saturation, and the third is high PCIe latency. To investigate these factors, we run the MEMSET workload on the desktop using both the regular DRAM speed (5600 MT/s) and a reduced speed (2400 MT/s). We use STREAM to measure our DRAM bandwidth using sufficiently large arrays to avoid cache effects as suggested by AMD [16]. The two DRAM speeds translate to an effective bandwidth of 38.16 GB/s and 22.11 GB/s, respectively, reducing the bandwidth by a factor of 1.73. For comparison, GDDR6 memory, which is used by all evaluated GPUs except the iGPU, has a bandwidth of up to 576 GB/s [14] on the RX 6950 XT and 224 GB/s [15] on the RX 6600. All dedicated GPUs are connected via PCIe 4.0 x16, which has a unidirectional bandwidth of 31.5 GB/s [61].

The BIOS of our desktop only supports a set of predefined DRAM speeds; among them, 2400 MT/s, which is 2.33 times slower than the regular speed, is the closest available option to half of the regular speed. Figure 27 shows our measurements using four configurations: dedicated GPU only with slow and regular DRAM speed, and both the dedicated and integrated GPUs also with slow and regular DRAM speed. Reducing the DRAM speed causes the tasks per second to reduce by a factor of 1.8 for the dGPU and 2.15 for the multi-GPU setting, which is close to the DRAM bandwidth reduction of 1.73. Thus, suggesting that the DRAM speed directly correlates with the tasks per second.

Furthermore, using the dGPU and iGPU together is more than twice as fast as using the dGPU alone, even though the iGPU has only a single worker compared to up to 80 workers on the dGPU. These results also suggest that DRAM is likely the bottleneck because the iGPU is not connected via PCIe; it likely uses different memory bandwidth resources than the dGPU and benefits from lower memory-access latency. Being physically closer to main memory and avoiding PCIe transfers allows the iGPU to contribute disproportionately to the overall performance. A similar effect was observed in the multi-GPU MEMSET workload shown in Figure 22, where using both GPUs nearly doubles per-worker performance, which also suggests distinct memory bandwidth resources for both devices.

Another notable observation emerges in the MEMSET workload using global memory on a remote GPU shown in Figure 17. All accesses to the remote GPU occur over PCIe via peer-to-peer DMA and are therefore subject to the same PCIe bandwidth limitations and PCIe latencies as accesses to host memory. Using remote global memory is between 2.11% slower at one worker and 120.97% slower at 80 workers compared to local global memory, yet still up to 114 times faster than using host memory. These measurements indicate that while using PCIe for all memory accesses can reduce performance, neither PCIe bandwidth nor latency is likely to be responsible for the severe performance loss observed when using host memory, as these factors would also affect the remote global memory setup.

Based on the results, we conclude that host memory bandwidth is likely the primary cause of the observed performance degradation. Additionally, different PCIe devices appear to have access to distinct DRAM bandwidth resources, which are likely allocated per PCIe root complex and depend on the total available DRAM bandwidth. The CPU not experiencing similar slowdowns is likely because it has access to the full DRAM bandwidth.

6.11. Discussion

The following discussion summarizes the findings of our evaluation and motivates several topics outlined in the future work section (Chapter 7). Additionally, we consolidate the findings and relate them to one another.

In this work we aimed to answer whether work stealing without CPU involvement is feasible. We firmly believe that our MGWS prototype indicates that even multi-device work stealing, including workers from multiple GPUs and the CPU, is possible in a decentralized manner without a CPU arbiter managing work on behalf of the GPU. While using host memory for inter-device communication represents a bottleneck, our evaluation indicates that using peer-to-peer DMA is a promising alternative.

We were unable to compare MGWS to existing CPU-managed work stealing systems for two main reasons. First, the source code of prior work was often not readily available. Second, most existing work relies on CUDA and is thus limited to NVIDIA GPUs. Our evaluation focuses on AMD GPUs, and a multi-GPU evaluation of MGWS is not possible on the NVIDIA GPUs available to us due to the lack of PCIe atomic support. While it is possible to convert CUDA to HIP using tools such as HIPIFY [17], doing so was beyond the scope of this thesis.

Nevertheless, we compare MGWS to a static task distribution, which relies on the CPU to assign each worker an equal number of tasks. Using a single GPU, we compare the static assignment to MGWS using global memory, global memory located on a remote GPU, and host memory for inter-worker communications. Due to hardware limitations, the multi-GPU evaluation is limited to the host memory-based inter-GPU communication scheme. However, using remote global memory in the single GPU environment allows us to approximate the performance of the peer-to-peer DMA-based multi-GPU communication scheme.

In the single-GPU setup under the MEMSET workload, direct task execution loading tasks from host memory performs similarly to using MGWS with global and remote global memory for inter-worker communications. In contrast, using host memory is up to 246 times slower than using global memory, as shown in Section 6.8.2. While this difference is less pronounced with the CONTAINS workload, particularly at lower worker counts, where the host memory configuration can outperform the global memory configuration. At high worker counts, the global memory configuration is up to 2.28 times faster than host memory, as discussed in Section 6.8.3. Under the CONTAINS workload, MGWS using global memory for inter-worker communications consistently outperforms the direct task execution.

In the multi-GPU setting under the regular workload, we observe the direct execution significantly outperforming MGWS, since without the peer-to-peer DMA-based scheme, inter-GPU communication must use host memory. Adding CPU workers, we observe MGWS outperforming direct task execution up to 48 workers, which suggests that MGWS is better suited to handle the uneven per-worker performance at low worker counts. Under the irregular workload, multi-GPU work stealing outperforms direct task execution for up to 96 workers, with the performance advantage being even greater when CPU workers participate.

The results indicate that MGWS is capable of outperforming direct task execution even when using the host memory-based communication scheme under the irregular workload or when CPU workers participate. These results hold for all but the highest worker counts, at which the overheads of the host memory scheme become too significant. Given that, in single-GPU evaluation, the remote global memory configuration outperforms the host memory approach in most cases, we would expect the peer-to-peer DMA-based scheme to achieve better performance in multi-GPU operations as well. The improvement from the peer-to-peer DMA-based scheme likely enables MGWS to outperform direct task execution in more scenarios. We leave it to future work to fully evaluate the peer-to-peer DMA-based inter-worker communication.

Our workloads were sufficient to examine the impact of irregular and regular tasks in addition to highlighting the overheads caused by different components of MGWS. However, future evaluation may gain new insights by utilizing additional workloads. Interesting workloads for future work include regular workloads with longer-running tasks, irregular workloads with shorter tasks, workloads that dynamically spawn tasks, and workloads that have a higher GPU affinity. When evaluating workloads with varying degrees of GPU or CPU affinity, modeling these affinities and taking them into account during work stealing may improve performance.

The significant performance difference between host memory and global memory-based communication, especially the large gap observed with the MEMSET workload, prompted us to investigate the causes for the substantial overhead incurred when using host memory. Since the performance degradation manifests on both test systems, we infer that it is not specific to our evaluating environment but instead reflects a more general limitation of host memory-based inter-device communication.

We initially identified three potential factors that may account for the large performance difference: PCIe bandwidth, PCIe latency, and host memory bandwidth. Using the remote global memory configurations, we were able to rule out PCIe bandwidth and PCIe latency as likely causes. If either of these factors were responsible, we would expect remote global memory and host memory to exhibit similar performance, since remote global memory also relies on the PCIe bus. However, under the MEMSET workload, using remote global memory is 114 times faster than using host memory, as shown in Section 6.8.2. Thus, the only remaining factor is the host memory bandwidth. To assess the impact of host memory bandwidth, we evaluate the MEMSET workload using two different host memory speeds, as shown in Figure 27. The measurements discussed in Section 6.10 show that reducing the effective memory bandwidth by a factor of 1.73 also decreases the performance of the dedicated GPU by a factor of 1.8. Based on these results, we conclude that the host memory bandwidth is the primary limiting factor.

Additionally, during the multi-GPU MEMSET evaluation shown in Figure 22, we observe that per-worker performance nearly doubles when both GPUs are used. Thus suggesting that both GPUs have distinct host memory bandwidth resources, which are likely assigned per PCIe root complex. We also believe that the host memory bandwidth allocated to each GPU scales with the overall memory bandwidth. However, further experiments are required to confirm this by directly measuring the bandwidth from different PCIe slots to the host with varying host memory speeds. A more in-depth understanding of the root cause of the bottleneck may enable future work to mitigate its effects, for instance, by reducing the volume of data transferred between each GPU and host memory. Some alternative approaches presented in Section 4.7 may be used for this purpose.

Using host memory is only one of our two implemented inter-GPU communication mechanisms. The second relies on peer-to-peer DMA, and while we were unable to evaluate it directly, we aim to approximate this communication scheme by accessing global memory located on the second GPU. Due to hardware limitations, only one GPU can access the remote global memory at the same time, thereby ensuring coherent memory accesses. In the remote global memory setup, all the MGWS orchestration data, including all public queues, resides in the global memory of the other GPU. Consequently, this setup is disadvantaged relative to the actual MGWS peer-to-peer DMA communication scheme, since in the latter, workers only access remote global memory when stealing tasks from workers on that remote GPU and not when accessing their own queues.

Nevertheless, based on the evaluation of both workloads, we expect that the peer-to-peer DMA-based communication approach will perform similarly to using local global memory, a scenario only possible in a single-GPU setting. Using remote global memory is up to 114 times faster than using host memory; see Section 6.8.2. Consequently, the peer-to-peer DMA communication scheme is likely superior to using host memory for inter-GPU communications, as it does not incur the same significant slowdown relative to local global memory. However, as our evaluation system demonstrates, support for peer-to-peer DMA does not necessarily imply the availability of coherent atomics over the connection. Furthermore, in its current implementation, MGWS does not support CPU worker participation when using the peer-to-peer DMA-based communications scheme. Enabling CPU worker participation while still leveraging the performance advantage of peer-to-peer DMA is therefore an interesting direction for future work.

Circumventing the PCIe bus for higher bandwidth and lower latency alternatives such as xGMI, NVLink or Compute Express Link (CXL) is also an interesting direction for future work and may enable even higher performance when using the peer-to-peer DMA-based scheme.

Evaluating the remote global memory performance under the CONTAINS workload provides not only insights into the expected performance of the peer-to-peer DMA-based communication scheme but also indicates the importance of inter-GPU data migration, which is currently not part of MGWS. In the remote global memory setup, the corpus is also located on the remote GPU, which is not the case for the local global and host memory configurations. The measurements reveal that, at least at high worker counts, inter-GPU data migration is likely less important since the remote global memory configuration outperforms the host memory configuration and is not far behind the local global memory configuration. Nevertheless, exploring inter-GPU data migration may still offer performance benefits depending on the workload, especially when direct access to foreign global memory via peer-to-peer DMA is unavailable.

In our multi-device evaluation, we not only compare MGWS to direct task execution but also both regular and irregular workloads under different restrictions on target selection, ranging from no restrictions to prohibiting stealing entirely. Restricting stealing under the host memory communication scheme does not provide any performance benefit, as all public queues reside in host memory; there is no difference between inter- and intra-device stealing. However, when using the peer-to-peer DMA-based approach, this is no longer the case. With this approach, stealing from a worker on the same device does not require device-to-device access and therefore incurs less overhead than stealing from a worker on a different device. Hence, when using peer-to-peer DMA, restricting stealing from workers on the other device reduces communication overhead at the cost of increased imbalance; however, the measurements show that for both workloads, the increased imbalance does not lead to significant performance loss.

Across all multi-device evaluations, the configuration that favors same-device workers consistently performs the best and is expected to also reduce communication overhead under the peer-to-peer DMA-based communication scheme, making it our preferred target selection strategy for MGWS.

Next, we highlight some findings related to specific parts of MGWS, starting with the evaluation of the termination detection and overflow prevention algorithms. In both scenarios, we compare approaches that are usually preferred in a CPU work stealing system against approaches that are simpler at the cost of likely higher contention on shared atomic counters.

In the case of the termination detection algorithms, the comparison is between a tree-based approach inspired by SWS [23] and a single atomic counter, which tracks the number of idle workers. The atomic counter-based approach exhibits consistently better performance when using global memory independent of the worker counts. Using host memory, both approaches perform nearly identically.

We observe similar results when comparing the overflow prevention algorithms that protect each public queue's attempted-steal counter, which is incremented using an atomic fetch-and-add as part of an attempted steal. The simplest approach to prevent overflowing the counter is to perform two atomic operations: first, checking whether work remains in the queue using an atomic fetch, and then, if work remains, incrementing the counter using an atomic fetch-and-add. While this approach prevents overflows, it is not as efficient as the steal damping algorithm proposed by Cartier et al. [23]. However, we observe no significant difference between the two approaches during our evaluation.

These two scenarios indicate optimizations aimed at reducing contention or the number of atomic operations targeting device or host memory are likely less impactful in GPU work stealing than in CPU work stealing. The highly optimized GPU atomics [49] are likely the main reason for this behavior. Overall, our findings demonstrate that strategies and optimizations commonly found in CPU work stealing are not always the best solution in GPU work stealing, and GPU-specific approaches can lead to performance advantages. Nevertheless, the tree-based approach, or steal dampening, may still be beneficial if the host memory is heavily contested using a high number of GPUs or if off-device accesses target shared storage via CXL. A hybrid approach between the counter and the tree may also be beneficial, where device internal termination detection uses the atomic counter and all devices combined use the tree-based approach. Alternatively, using a hierarchical counter approach may also avoid significant contention on the global counter if a high number of GPUs are used.

Our evaluation of multiple queue sizes and public queue ownership sharing revealed that some memory configurations perform better with smaller queues or with ownership sharing. We did not expect ownership sharing to outperform configurations without sharing, since it was originally implemented solely to reduce the memory footprint of MGWS. Similarly, we initially assumed that larger queues would always improve performance, as more tasks could be stolen or loaded from the initial task set at once. However, the findings demonstrate that these assumptions were incorrect. Consequently, careful tuning of parameters for specific inter-worker communication mechanisms and worker counts can be beneficial, as the same configuration may perform differently depending on the environment.

By default, we recommend using task queues with 64 elements and no ownership sharing, since this configuration achieves the best performance at high worker counts across all communication schemes. The only exception is using global memory, where the performance difference compared to larger queues is negligible.

In conclusion, MGWS demonstrates that work stealing without CPU orchestration is feasible. Furthermore, our evaluation highlights the critical role of the inter-GPU communication mechanism for overall performance, even under regular workloads, and suggests that further development of MGWS and evaluation of the peer-to-peer DMA scheme could provide additional insights into this topic.

7. Future Work

The current implementation of MGWS still offers interesting directions for future extension and evaluation. Potential extensions include task modeling to capture dependencies and GPU-CPU affinity, enabling CPU participation in work stealing even when peer-to-peer DMA is used, inter-GPU data migration, and approaches to mitigate the host memory bottleneck. Future evaluation may include the peer-to-peer DMA-based inter-GPU communication scheme, additional workloads, GPU vendors, and GPU interconnects.

CPU participation when using peer-to-peer DMA: In MGWS, the CPU cannot directly access global memory. As a result, when utilizing our peer-to-peer DMA-based communication scheme, the CPU cannot participate in work stealing, or only one-directional steals from the GPU to the CPU are possible since the public task queues reside in global memory. We identify two potential directions for future work to eliminate this limitation. First, a Field-Programmable Gate Array (FPGA) could participate in peer-to-peer DMA on behalf of the CPU, which should be feasible, as the Input-Output Memory Management Unit (IOMMU) of modern GPUs allows peer-to-peer DMA with other PCIe devices [18]. Second, a simpler alternative is to rely on resizable Base Address Register (BAR). However, this approach would require additional changes to MGWS since, to the best of our knowledge, atomic operations are not supported over resizable BAR.

Inter-GPU data migration: MGWS already includes data descriptors as part of the task struct but offers no explicit mechanism for migrating or replicating task data. Implementing our own data migration algorithm based on heuristics to decide whether a given migration is beneficial to minimize overall execution time may allow better data management compared to ROCm HIP built-in solutions such as unified memory or registered memory. Works such as [20,32,38,42] provide intriguing ideas for inter-GPU data management, including versioning data, letting users provide data transfer functions, or maintaining a software cache.

Model task dependencies: The only way to model task dependencies in MGWS is by letting tasks spawn new subtasks. However, more sophisticated task dependencies, such as task dependency graphs as seen in [25,38,42] are often desired. ROCm HIP provides *HIP graphs* [19] as part of their API, which allows developers to express dependencies between operations, including kernel launches, host functions, and HIP memory functions. It would be interesting to see if HIP graphs or a similar approach could be adopted to work with MGWS. D3D12 Work Graphs [62] is another dependency-graph approach designed for producer-consumer workloads and aims to support dynamic resource allocation for tasks.

Another approach to model dependencies is to let tasks wait for other tasks or conditions, which could be implemented by introducing a global registry of waiting conditions and providing each worker with an additional buffer for waiting tasks. The global registry may be realized using a hash table. In this design a worker can periodically check, between task executions, whether it has waiting tasks and whether a waiting condition of any such task has been satisfied, either by the worker itself or by another worker. These checks may be performed in parallel, and depending on the number of waiting tasks, a round-robin strategy may be used to select a subset of the waiting tasks to examine.

Model GPU and CPU affinity of tasks: Existing solutions such as [20] allow the user to explicitly specify whether a task prefers to be executed on the CPU or GPU, while MGWS supports sharing tasks between GPU and CPU workers; we have no influence over which worker ultimately executes a given task. Without a centralized scheduler, which we want to avoid, ensuring that tasks stay on their preferred worker type is challenging. However, task affinity to CPU or GPU workers can be preserved when loading the initial task set. Subsequently, CPU and GPU workers may prefer stealing from the same type of worker as long as the system maintains sufficient remaining work. Such a preference would at least increase the likelihood that a task is executed by its preferred worker type while maintaining our decentralized design.

A more involved solution could additionally assign each task a tag specifying whether they prefer GPU or CPU workers. The public queue could then store CPU tasks on one end and GPU tasks on the other end, allowing steal operations targeting both ends. Usually this would be difficult to realize because one end of the task queue is often used by the queue owner to enqueue new tasks. However, because MGWS uses two separate queues and locks the public queue when it acquires or releases new tasks, this is not an issue. That said, extending the metadata to contain information about both queue ends while still allowing lock-free stealing may be difficult.

Intercept kernel launches: Intercepting kernel launches and then mapping the invocation to tasks, which can then be distributed by MGWS, as seen in [38], may make it easier for existing applications to adopt MGWS for load balancing. Moreover, HIP memory functions could also be intercepted and replaced with our own inter-GPU data management scheme.

Explore new ways to circumvent the host memory bottleneck: As discussed in our evaluation, storing public queues in host memory leads to bottlenecks and limits the number of tasks that can be executed in a given time. An alternative to reduce the data that needs to be transferred between queues would be to represent each task instance by a unique ID, as demonstrated by [41]. Thus allowing the storage of IDs in public queues instead of the real tasks, thereby reducing the amount of data that needs to be transferred between queues. When a worker acquires new tasks from its public queue or enqueues new tasks in its local queue, the real tasks may be stored instead to reduce overhead when executing the tasks. These tasks are likely to stay in the local queue and are thus not moved anymore.

It would also be interesting to see if the other approaches we presented in Section 4.7 can elevate this bottleneck. The first approach discussed uses a global task pool and reservation flags, and the second models task ownership similar to a cache consistency problem with MOESI.

Evaluate the peer-to-peer DMA-based communication scheme: In this work, we approximated the peer-to-peer DMA-based inter-GPU communication scheme by using peer-to-peer DMA to access memory on a remote GPU. However, we were unable to directly evaluate the full communication mechanism, as it requires coherent atomic operations over peer-to-peer DMA, which were not supported by our evaluation setup. Switching PCIe slots in the evaluation system might resolve this limitation, but the physical size of the GPUs prevented us from testing alternative slot configurations. Future work should therefore evaluate this communication scheme on a system with adjacent PCIe slots that support atomics over a peer-to-peer connection.

Evaluate NVIDIA GPUs: MGWS is developed using ROCm HIP, which makes it possible to run on both AMD and NVIDIA GPUs, and it would be especially interesting to evaluate work stealing between AMD and NVIDIA GPUs. We tested our code on NVIDIA GPUs in a single-GPU setting. However, we did not have access to NVIDIA GPUs supporting PCIe atomics, which makes testing multiple NVIDIA GPUs or a combination of NVIDIA and AMD GPUs impossible. An alternative to relying on PCIe atomics would be to use NVLink but we also did not have access to NVLink bridges or compatible NVIDIA GPUs.

Evaluate additional interconnects: Theoretically, MGWS supports NVLink and xGMI without any modifications since these special interconnects use the same API as PCIe peer-to-peer DMA. As an alternative to the interconnects of AMD and NVIDIA, CXL may also be used for GPU-to-GPU connections or to connect to fast shared storage. We are intrigued to compare performance characteristics between PCIe-based peer-to-peer DMA and the faster GPU interconnects. However, we did not have access to either NVLink or xGMI-compatible GPUs.

Evaluate additional workloads: It would be interesting to evaluate MGWS on a broader range of workloads, particularly real-world irregular workloads that dynamically spawn new tasks, such as graph traversal. The irregular workload we evaluated does not spawn new tasks at runtime and executes faster on the CPU than the GPU. While the CONTAINS workload was sufficient for an initial evaluation, additional workload may provide further insights.

Promising workload categories include regular workloads with longer-running tasks, irregular workloads with shorter tasks, workloads that dynamically spawn tasks, and workloads that have a higher GPU affinity.

8. Conclusion

This work presents MGWS, a novel multi-GPU work stealing algorithm designed to enable GPU workers to operate independently of the CPU, thereby reducing communication and synchronization overheads incurred through CPU involvement. Furthermore, our approach avoids any single worker or scheduler becoming a bottleneck by adopting a decentralized design, in which each worker autonomously steals work on its own.

To this end, we propose two inter-GPU communication schemes: the first is based on host memory mapped to all GPUs, while the second relies on peer-to-peer DMA for direct GPU-to-GPU communication. In addition to GPU workers, MGWS also supports CPU workers using the same code base. The host memory-based approach theoretically enables the use of GPUs from different vendors simultaneously; however, due to hardware limitations, we were unable to evaluate this capability.

Our evaluation explores multiple variations of MGWS, including two distinct termination detection algorithms. The results indicate that a simple atomic counter-based approach can match or even outperform a more sophisticated tree-based approach by up to 73.54%. We further find that, depending on the memory type used to store the public queues and the worker count, smaller queues and ownership-sharing of public queues can outperform larger queues without ownership-sharing.

A key finding of our evaluation is the substantial performance impact of the inter-worker communication mechanism. The host memory-based approach is up to 246 times slower than using global memory for short regular tasks, motivating an investigation into the causes of this overhead and potential mitigation strategies. The performance gap is less pronounced when executing irregular tasks that dominate the total execution time; in this case, host memory is on average at most 2.28 times slower than global memory. Due to hardware limitations, we were unable to directly evaluate the peer-to-peer DMA-based communication scheme. Instead, we approximate this approach in a single GPU setting by allowing a GPU to access global memory on a remote GPU using peer-to-peer DMA. Measurements using remote global memory for inter-worker communication are promising and do not exhibit the severe performance gap observed with the host memory-based approach, increasing our confidence that the peer-to-peer DMA-based inter-GPU communication mechanism can achieve near-global-memory performance.

Overall, this work motivates further extension and evaluation of MGWS. In particular, future work includes validating the performance of the peer-to-peer DMA-based communication scheme on suitable hardware and evaluating MGWS using real-world workloads with higher GPU affinity. Exploring circumventing the PCIe bus entirely through xGMI, NVLink, or CXL is another interesting direction. Finally, investigation designs to mitigate the overhead incurred by host memory-based communication remain an intriguing avenue for future research.

Bibliography

1. TOP500 November 2024. Retrieved from https://www.top500.org/lists/top500/2024/11/
2. Instruction Latency. Retrieved from https://forums.developer.nvidia.com/t/instruction-latency/3579/8
3. Wortschatz-Portal der Universität Leipzig "zwischen". Retrieved from https://dict.wortschatz-leipzig.de/de/res?corpusId=deu_news_2024&word=zwischen
4. AMD. HIP. Retrieved from https://github.com/ROCm/HIP
5. AMD. HIP Hardware Implementation — HIP Documentation. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/understand/hardware_implementation.html
6. AMD. AMD CDNA™ 3 ARCHITECTURE. Retrieved from https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf
7. AMD. AMD Memory Management. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___memory.html#gab8258f051e1a1f7385f794a15300e674
8. AMD. HIP Programming Model — HIP Documentation. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html
9. AMD. Understanding RCCL Bandwidth and xGMI Performance on AMD Instinct™ MI300X. Retrieved from https://rocm.blogs.amd.com/software-tools-optimization/mi300x-rccl-xgmi/README.html
10. AMD. Multi Device Peer-to-peer memory access — HIP Documentation. Retrieved from https://rocmdocs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/multi_device.html#peer-to-peer-memory-access
11. AMD. How ROCm uses PCIe atomics. Retrieved from https://rocm.docs.amd.com/en/docs-6.3.1/conceptual/pcie-atomics.html#pcie-for-atomic-operations
12. AMD. HIP API Stream Management. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___stream.html#ga3076a3499ed2c7821311006100bb95ec
13. AMD. hip-libraries-rocm-ubuntu.Dockerfile. Retrieved from https://github.com/amd/rocm-examples/blob/develop/Dockerfiles/hip-libraries-rocm-ubuntu.Dockerfile
14. AMD. Radeon™ RX 6950 XT Desktop Graphics Card. Retrieved from https://www.amd.com/en/products/graphics/desktops/radeon/6000-series/amd-radeon-rx-6950-xt.html
15. AMD. Radeon™ RX 6600 Graphics Card. Retrieved from https://www.amd.com/en/products/graphics/desktops/radeon/6000-series/amd-radeon-rx-6600.html
16. AMD. STREAM Benchmark. Retrieved from https://www.amd.com/en/developer/zen-software-studio/applications/spack/stream-benchmark.html
17. AMD. HIPIFY documentation. Retrieved from https://rocm.docs.amd.com/projects/HIPIFY/en/latest/index.html
18. AMD. Input-Output Memory Management Unit (IOMMU). Retrieved from https://instinct.docs.amd.com/projects/amdgpu-docs/en/latest/conceptual/iommu.html
19. AMD. HIP graphs. Retrieved from https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/hipgraph.html

20. Humayun Arafat, James Dinan, Sriram Krishnamoorthy, Pavan Balaji, and P Sadayappan. 2016. Work stealing for GPU-accelerated parallel programs in a global address space framework. *Concurrency and Computation: Practice and Experience* 28, 13: 3637–3654.
21. Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, 119–129.
22. Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5: 720–748.
23. Hannah Cartier, James Dinan, and D. Brian Larkins. 2021. Optimizing Work Stealing Communication with Structured Atomic Operations. In *Proceedings of the 50th International Conference on Parallel Processing* (ICPP '21). https://doi.org/10.1145/3472456.3472522
24. Daniel Cederman and Philippas Tsigas. 2012. Dynamic load balancing using work-stealing. *GPU Computing Gems Jade Edition*, 485–499.
25. Sanjay Chatterjee, Max Grossman, Alina Sb\irlea, and Vivek Sarkar. 2011. Dynamic task parallelism with a GPU work-stealing runtime system. In *International Workshop on Languages and Compilers for Parallel Computing*, 203–217.
26. Bhuvi Chopra. 2024. Enhancing Machine Learning Performance: The Role of GPU-Based AI Compute Architectures. *J. Knowl. Learn. Sci. Technol* 6386: 29–42.
27. Ching-Hsiang Chu, Sreeram Potluri, Anshuman Goswami, Manjunath Gorentla Venkata, Neena Imam, and Chris J Newburn. 2019. Designing high-performance in-memory key-value operations with persistent gpu kernels and openshmem. In *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity: 5th Workshop, OpenSHMEM 2018, Baltimore, MD, USA, August 21–23, 2018, Revised Selected Papers 5*, 148–164.
28. James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 1–11.
29. Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (ISCA '11), 365–376. https://doi.org/10.1145/2000064.2000108
30. Aleksandra Franz, Hao Wei, Luca Guastoni, and Nils Thuerey. 2025. PICT–A Differentiable, GPU-Accelerated Multi-Block PISO Solver for Simulation-Coupled Learning Tasks in Fluid Dynamics. *arXiv preprint arXiv:2505.16992.*
31. Esraa A Gad. 2017. A Work-Stealing For Dynamic Workload Balancing On CPU-GPU Heterogeneous Computing Platforms.
32. Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. 2013. Locality-aware work stealing on multi-CPU and multi-GPU architectures. In *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG).*
33. Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. 2017. IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems. *Concurrency and Computation: Practice and Experience* 29, 9: e4019.
34. Weiwei Gong and Xu Zhou. 2017. A survey of SAT solver. In *AIP Conference Proceedings*, 20059.
35. The Khronos® Group. OpenCL Overview. Retrieved from https://www.khronos.org/opencl/
36. The Khronos® Group. Vulkan. Retrieved from https://www.vulkan.org/

37. John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
38. Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. 2010. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part II 16*, 235–246.
39. Sungjin Im, Ravi Kumar, Silvio Lattanzi, Benjamin Moseley, Sergei Vassilvitskii, and others. 2023. Massively parallel computation: Algorithms and applications. *Foundations and Trends® in Optimization* 5, 4: 340–417.
40. Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*.
41. Joseph John and Josh Milthorpe. 2024. Elasticity in a Task-based Dataflow Runtime Through Inter-node GPU Work Stealing. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 97–105.
42. Joseph John, Josh Milthorpe, Thomas Herault, and George Bosilca. 2024. Multi-GPU work sharing in a task-based dataflow programming model. *Future Generation Computer Systems* 156: 313–324.
43. Khronos. The OpenCL™ Specification. Retrieved from https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_the_opencl_architecture
44. Suhwan Kim, Changue Jung, and Younghoon Kim. 2022. Comparative Analysis of GPU Stream Processing between Persistent and Non-persistent Kernels. In *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, 2330–2332. https://doi.org/10.1109/ICTC55196.2022.9952789
45. MA Lastras-Montano, MM Michael, and JA Bivens. 2010. Dynamic work scheduling for gpu systems. In *International Workshop of GPUs and Scientific Applications, GPUScA*.
46. Joao VF Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. 2012. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 75–82.
47. Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 257–270.
48. Vincent Magnoux and Beno\'it Ozell. 2021. GPU-friendly data structures for real time simulation. *Advanced modeling and simulation in engineering sciences* 8, 1: 7.
49. Peter Maucher, Nick Djerfi, Lennard Kittner, Lukas Werling, and Frank Bellosa. 2025. Are Your GPU Atomics Secretly Contending?. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems* (PLOS '25), 84–92. https://doi.org/10.1145/3764860.3768338
50. Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, Lukas Werling, Yussuf Khalil, Thorsten Gröninger, and Frank Bellosa. 2024. Full-Scale File System Acceleration on GPU. In *Tagungsband des FG-BS Frühjahrstreffens 2024*, 10–18420.
51. Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. 2023. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 191–204.
52. Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2. https://doi.org/10.1145/2717511

53. NASA. Basics on NVIDIA GPU Hardware Architecture. Retrieved from https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html
54. NVIDIA. CUDA Zone. Retrieved from https://developer.nvidia.com/cuda-zone
55. NVIDIA. Device Memory Accesses. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses
56. NVIDIA. Unlock GPU Performance: Global Memory Access in CUDA. Retrieved from https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/
57. NVIDIA. CUDA Memory Management. Retrieved from https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8d5c17670f16ac4fc8fcb4181cb490c
58. NVIDIA. CUDA C++ Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
59. NVIDIA. NVIDIA NVLink and NVLink Switch. Retrieved from https://www.nvidia.com/en-us/data-center/nvlink/
60. NVIDIA. Peer-to-Peer & Unified Virtual Addressing. Retrieved from https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf
61. Kevin Parrish. PCI Express 4.0: What it is and why it's important. Retrieved from https://www.androidauthority.com/pci-express-4-0-995998/
62. Amar Patel and Tex Riddell. D3D12 Work Graphs. Retrieved from https://devblogs.microsoft.com/directx/d3d12-work-graphs/
63. Nick Curtis, Chip Freitag, Joe Greathouse, Nicholas Malaya, Damon McDougall, Scott Moe, René van Oostrum, Noah Wolfe Paul Bauman Noel Chalmers. 2019. INTRODUCTION TO AMD GPU PROGRAMMING WITH HIP. Retrieved from https://www.olcf.ornl.gov/wp-content/uploads/2019/09/AMD_GPU_HIP_training_20190906.pdf
64. Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 1–8.
65. Julio Toss. 2011. Work stealing inside GPUs.
66. Stanley Tzeng, Anjul Patney, and John D Owens. 2010. Task management for irregular-parallel workloads on the GPU.

Appendix

A Memory Latency Measurement Code

```
#include "hip/hip_runtime.h"
#include <cstddef>
#include <cstdint>
#include <cstdio>
#include <iostream>

const size_t ALLOCATION_SIZE            = 17160000000 / 4;
const size_t SHARED_MEM_ALLOCATION_SIZE = 65536 / 2;
const int ITERATIONS                    = 1000;
const size_t SHARED_MEM_STRIDE = SHARED_MEM_ALLOCATION_SIZE / sizeof(uint64_t) / ITERATIONS;
const size_t STRIDE            = ALLOCATION_SIZE / sizeof(uint64_t) / ITERATIONS;
const int SKIP                 = 1;

__device__ void print_measurements(uint64_t *mem, uint64_t stride) {
    if (threadIdx.x == 0) {
        uint64_t min_val = -1LLU;
        uint64_t max_val = 0;
        uint64_t avg_val = 0;
        for (int i = 0; i < ITERATIONS; i++) {
            uint64_t start  = clock64();
            mem[stride * i] = 123;
            __threadfence_system();
            uint64_t end = clock64();
            if (i >= SKIP && start < end) {
                min_val = min(min_val, end - start);
                max_val = max(max_val, end - start);
                avg_val += end - start;
            } else {
                printf("Skipped measurement took %lu\n", end - start);
            }
        }
        printf("Measurement took min %lu avg %lu max %lu\n", min_val, avg_val / ITERATIONS, max_val);
    }
    __syncthreads();
}

__global__ void test_latency(uint64_t *registered_memory, uint64_t *unified_memory, uint64_t *global_memory) {
    __shared__ uint64_t shared_memory[SHARED_MEM_ALLOCATION_SIZE / sizeof(uint64_t)];
    print_measurements(shared_memory, SHARED_MEM_STRIDE);
    print_measurements(global_memory, STRIDE);
    print_measurements(registered_memory, STRIDE);
    print_measurements(unified_memory, STRIDE);
}

int main(int argc, char **argv) {
    uint64_t *host_mem = reinterpret_cast<uint64_t *>(malloc(ALLOCATION_SIZE));
    HIP_CHECK(hipHostRegister(host_mem, ALLOCATION_SIZE, 0));
    uint64_t *registered_mem;
    HIP_CHECK(hipHostGetDevicePointer(reinterpret_cast<void **>(&registered_mem), host_mem, 0));
    uint64_t *global_mem;
    HIP_CHECK(hipMalloc(&global_mem, ALLOCATION_SIZE));
    uint64_t *unified_mem;
    HIP_CHECK(hipMallocManaged(&unified_mem, ALLOCATION_SIZE));
    for (int i = 0; i < 50; i++) { unified_mem[STRIDE * i] = 42; }
    hipLaunchKernelGGL(test_latency, 1, 128, 0, nullptr, registered_mem, unified_mem, global_mem);
    HIP_CHECK(hipGetLastError());
    HIP_CHECK(hipDeviceSynchronize());
    HIP_CHECK(hipFree(unified_mem));
    HIP_CHECK(hipFree(global_mem));
    HIP_CHECK(hipHostUnregister(host_mem));
    free(host_mem);
}
```

Listing A8 : The program measures write access latencies for shared, global, registered, and unified memory. We report latencies in clock cycles using the HIP function `clock64`. The program reports the minimum, maximum, and average access times while excluding the first measurement, which is usually an outlier, and any measurement impacted by overflows of the clock counter, which only happens once or twice per program execution. We also use a stride to reduce cache effects.

B Test Stream Destruction behavior

```
#include "hip/hip_runtime.h"
#include <cstdint>
#include <cstdio>
#include <iostream>
#include <atomic>

__global__ void print(uint64_t j, uint64_t *continue_execution, uint64_t *result_area) {
    // there is no atomicLoad
    if (threadIdx.x == 0)
        while (atomicAdd(continue_execution, 0) == 0);
    __syncthreads();
    if (threadIdx.x == 0) atomicAdd(result_area + j, 1LU);
    __syncthreads();
}

const uint64_t NUM_KERNELS = 4096;

int main(int argc, char **argv) {
    hipStream_t stream;
    uint64_t *result_area;
    HIP_CHECK(hipMallocManaged(&result_area, sizeof(uint64_t) * NUM_KERNELS));
    uint64_t *continue_execution;
    HIP_CHECK(hipMallocManaged(&continue_execution, sizeof(uint64_t)));
    auto *continue_execution_atomic_ptr =
      reinterpret_cast<std::atomic<uint64_t> *>(continue_execution);
    continue_execution_atomic_ptr->store(0);
    for (int i = 0; i < NUM_KERNELS; i++) { result_area[i] = 0; }

    HIP_CHECK(hipStreamCreate(&stream));
    for (uint64_t i = 0; i < NUM_KERNELS; i++) {
        hipLaunchKernelGGL(print, 1, 128, 0, nullptr, i, continue_execution,
          result_area);
        HIP_CHECK(hipGetLastError());
    }
    HIP_CHECK(hipStreamDestroy(stream));
    std::cout << "STREAM DESTROYED" << std::endl;
    for (int i = 0; i < NUM_KERNELS; i++) {
        auto *atomic_ptr = reinterpret_cast<std::atomic<uint64_t> *>(result_area + i);
        assert(atomic_ptr->load() == 0);
    }

    continue_execution_atomic_ptr->store(1);
    HIP_CHECK(hipDeviceSynchronize());
    uint64_t kernels_finished = 0;
    for (int i = 0; i < NUM_KERNELS; i++) {
        auto *atomic_ptr = reinterpret_cast<std::atomic<uint64_t>
        *>(result_area + i);
        kernels_finished += atomic_ptr->load();
    }
    std::cout << "Executed: " << kernels_finished << " Submitted: " <<
    NUM_KERNELS << std::endl;
    HIP_CHECK(hipFree(result_area));
    HIP_CHECK(hipFree(continue_execution));
}
```

Listing B9 : The program tests whether already submitted kernels are still executed even after the stream they were submitted to is destroyed. To test this, we submit kernels to a stream and destroy that stream before the kernels have been executed. Kernels submitted to the same stream are executed in series, one at a time. The first kernel spins to delay further execution until after the stream was destroyed. Next, each kernel writes into host-accessible memory. Finally, the host counts how many kernels were able to write a value.

C Test Peer-To-Peer Atomics

```
#include "hip/hip_runtime.h"
#include <cstdint>
#include <cstdio>
#include <iostream>
#include <string>

__global__ void write(uint64_t *value) {
    __threadfence_system();
    printf("read: %lu\n", atomicAdd_system(value, 1LU));
    __threadfence_system();
}

__global__ void spin(uint64_t *value) {
    // there is no atomicLoad_system
    while (NUMBER_OF_LANES > atomicAdd_system(value, 0LU)) { __threadfence_system(); }
    printf("done\n");
}

int main(int argc, char **argv) {
    int out;
    HIP_CHECK(hipDeviceCanAccessPeer(&out, 0, 1));
    std::cout << out << std::endl;
    HIP_CHECK(hipDeviceCanAccessPeer(&out, 1, 0));
    std::cout << out << std::endl;

    HIP_CHECK(hipDeviceEnablePeerAccess(1, 0));
    HIP_CHECK(hipSetDevice(1));
    HIP_CHECK(hipDeviceEnablePeerAccess(0, 0));

    HIP_CHECK(hipSetDevice(0));
    uint64_t *ptr_gpu_0 = nullptr;
    uint64_t zero       = 0;
    HIP_CHECK(hipMalloc(&ptr_gpu_0, sizeof(uint64_t)));
    HIP_CHECK(hipMemcpy(ptr_gpu_0, &zero, sizeof(uint64_t), hipMemcpyHostToDevice));
    HIP_CHECK(hipDeviceSynchronize());

    hipLaunchKernelGGL(spin, 1, NUMBER_OF_LANES, 0, nullptr, ptr_gpu_0);
    HIP_CHECK(hipGetLastError());
    HIP_CHECK(hipSetDevice(1));
    hipLaunchKernelGGL(write, 1, NUMBER_OF_LANES, 0, nullptr, ptr_gpu_0);
    HIP_CHECK(hipGetLastError());

    HIP_CHECK(hipDeviceSynchronize());
    HIP_CHECK(hipSetDevice(0));
    HIP_CHECK(hipDeviceSynchronize());
    HIP_CHECK(hipFree(ptr_gpu_0));
}
```

Listing C10 : In this program we simulate a spin lock via peer-to-peer DMA to test whether atomic operations perform as expected. One GPU starts spinning while the other atomically sets a variable that can be accessed by both GPUs via peer-to-peer DMA.

D Replicate Evaluation

This section explains how our evaluation can be reproduced. Any system used to reproduce the results must have two GPUs with at least as many compute units as the RX 6950 XT and RX 6600, since our benchmarks are tuned for our system.

D Create Wikipedia Data

Generating the Wikipedia data requires Rust.

```
wget --show-progress https://dumps.wikimedia.org/dewiki/20251120/dewiki-20251120-pages-articles-multistream.xml.bz2
bzip2 -d dewiki-20251120-pages-articles-multistream.xml.bz2
git clone https://github.com/LennardKittner/wikipedia_extractor.git
cd wikipedia_extractor
cargo run --release -- -p ../dewiki-20251120-pages-articles-multistream.xml -n 100000 -o /tmp/wiki_100000 --output-format=single-file-with-index
```

D Build MGWS

We assume that Google Test and HIP are already installed. Alternatively, we provide a Docker-File which comes with both pre-installed. MGWS may not be immediately publicly available; however, once it is released, the following steps can be taken to build MGWS.

```
git clone https://github.com/LennardKittner/MGWS.git
cd mgws
rm -rf build && cmake -B build && cmake --build build -j
```

The README.md provides additional information on how MGWS can be built for NVIDIA GPUs or in CPU-only mode.

D Reproduce The Results

`./run_benchmarks.sh` inside the root of the repository executes all benchmarks and writes the results into `data`. The script only repeats each benchmark five times; thus, results with 200 repetitions cannot be generated automatically. Generating the results may take a few hours. Every benchmark generates a file with information about the build configuration, as well as the average, minimum, and maximum execution times in seconds.

D Display The Results

This thesis document may not be immediately publicly available; however, once it is released, the following steps can be taken to build the document with the new measurements.

```
git clone https://github.com/LennardKittner/MGWS_thesis.git
```

Merge the `data` folder generated using `./run_benchmarks.sh` and the `data` folder inside the `MGWS_thesis`. To compile the document, Typst is required.

```
typst compile thesis.typ && open thesis.pdf
```