On Heterogeneous Systems and Data Repositories

by

dePaul Miller

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

January 2025

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____
Date

_____
Dissertation Advisor

Committee Members:

_____
Roberto Palmieri, Committee Chair

_____
Ahmed Hassan

_____
Lewis Tseng

_____
Hank Korth

_____
Mike Spear

Acknowledgements

I want to acknowledge my immediate and extended family's support throughout this journey. My parents, Cynthia and Jeffrey, and my brothers, Jeffrey and Ronan, have been incredibly supportive throughout my academic journey. They have always encouraged me to pursue my educational goals. I would especially like to recognize those who are no longer with us, especially my Aunt Stacy. She inspired me to pursue science and higher education by constantly conversing about my current scientific interests and taking me to some of her STEM education conferences.

I would also like to acknowledge the support of the Scalable Systems and Software lab at Lehigh University. Masoomeh Javidi Kishi, Pantea Zardoshti, Jacob Nelson-Slivon, Matthew Rodriguez, Yaodong Sheng, Amanda Baran, Olivia Grimes, Reilly Yankovich, Rishad Islam, Sheldon Xu, and Alex Clevenger have taught me a lot throughout this journey, indulged me in talking about heterogeneous computation, and made the journey more enjoyable with our lab outings. I would especially like to acknowledge the professors who helped shape how I approach research, presentations, and academics, including Professors Roberto Palmieri, Ahmed Hassan, Hank Korth, Mike Spear, and Arielle Carr.

Finally, I would like to acknowledge all of my committee members, Professors Roberto Palmieri, Ahmed Hassan, Hank Korth, Mike Spear, and Lewis Tseng, for their invaluable mentorship throughout this journey on topics ranging from presentations to data structures and formal theory and verification. I owe special thanks to Professors Ahmed Hassan and Hank Korth for their collaboration and input throughout my PhD. I would especially like to thank my advisor and committee chair, Roberto Palmieri, for his invaluable mentorship during my undergraduate and doctorate studies.

Table of Contents

List of Tables

List of Figures

Abstract

Data repositories are software systems that store, retrieve, and analyze data. These repositories are the backbone of computing infrastructure and rely on various core components, including concurrency controls and data structures. It is essential to improve the performance of these systems to support ever-increasing computational demand. With recent trends in computer architecture, it is becoming increasingly important to consider specialized processors and how they are interconnected and can cooperate. The design of these heterogeneous systems for data repositories and their algorithmic components is the primary focus of this dissertation. More specifically, we consider utilizing central processing units (CPUs) with graphics processing units (GPUs) as co-processors and designing our systems and components with these processors in mind. Our methodology is to approach data repositories through the lens of *instruction set architecture affinity* (ISA affinity), or how well our algorithms and tasks map to specific processor architectures. We further consider the interconnection between processors and the additional latency and performance of moving data between co-processors. The contributions of this dissertation include the design of two systems: a cooperative CPU-GPU key-value store and a transactional system with support for heterogeneous workloads, including hybrid transactional-analytical processing through first-class support for heterogeneous architectures. We also provide an approach to semantic transactional processing through cooperative CPU-GPU processing, an architecture-agnostic framework for coalescing memory accesses in data structures for high performance, and a mapping data structure supporting linearizable point operations and range queries.

Chapter 1

Introduction

Moore's law has been arguably the most impactful law governing computing since the 1960s. The law, attributed to Gordon Moore, co-founder of Intel, posits that every 18 months, the density of transistors in a semiconductor will double [124]. For a long time, this held with another law of computing, Dennard scaling [52], the idea that as transistors get smaller, their power density stays near constant. Anyone using a computer could expect their processor's frequency to scale accordingly, and upgrading processors would result in significant speed improvements.

Over time, it became evident that Dennard scaling would no longer hold [62], and chip-makers moved to improve performance by introducing increasing processor cores. Within the past 20 years, there has been a focus on developing scalable software to take advantage of increasing core counts in central processing units (CPUs).

Concurrently with this multicore revolution, there have been many computer architectural developments to target and accelerate specific problems, which have similarly benefitted from Moore's Law and Dennard Scaling. As early as the 1980s, James H. Clark was working on the geometry engine [43], a special-purpose processor for 3D graphics. This architecture was later used as the basis for the graphics processing workstations released by Silicon Graphics.

In 1993, NVIDIA was created as a graphics processor unit (GPU) company targeting video games and multimedia acceleration on personal computers. NVIDIA's target application of their GPUs has continued to evolve to general-purpose computing. In 2003, Ian

Buck et al. published Brook [32], an extension to the C programming languages and a library that presented the GPU as a streaming coprocessor to the CPU with data parallelism. These ideas later made their way into the CUDA framework when released in 2006 [142]. This framework has proven useful, especially for using the GPU as a coprocessor with the CPU for deep learning and linear algebra.

We are now in an era with extreme demand for computation, while Moore's Law has seemingly slowed. Processors are fabricated at reticle limit sizes [111], and solutions such as multi-chip modules [17] and die stacking [112] are heavily explored in academia and industry as techniques to have processors with as many transistors as possible. These solutions lead to non-uniform cache accesses (NUCA) and non-uniform memory accesses (NUMA), where there are different latencies for accessing different cache lines or memory regions, depending on which die it is connected to.

Within this landscape, computer architects, software systems engineers, and researchers must explore a variety of design tradeoffs and paths toward optimized computation. These decisions can be made within specific systems and, more broadly, as standard optimizations that yield optimal results across architectures.

The most fundamental approach to this problem we explore in this dissertation is understanding *instruction set architecture (ISA) affinity*. ISA affinity is the relationship between a task or algorithm and the processor on which it performs best. Evaluating ISA affinity alone is insufficient; we must also understand how processors within our system architecture are interconnected.

We often interconnect processors through networks such as UCIe, PCIe, CXL, Ethernet, Infiniband, etc. Interconnection networks add latency to communication between processors, which affects system performance. Furthermore, networks have implications for essential properties of software systems, such as safety and liveness. Networks such as Infiniband provide unique primitives that can be utilized to develop high-performance and scalable software. Infiniband introduces remote direct memory access (RDMA) primitives. These primitives can bypass the operating system kernel to communicate between computers and perform atomic operations on memory on a remote processor. To develop future software systems, researchers and designers must have a firm grasp of the architecture of

the systems available to them and the underlying problem they wish to solve.

In this dissertation, we evaluate systems and components of *data repositories* through the lens ISA affinity of CPU and GPU architectures. Data repositories are software systems that provide the ability for clients to retrieve, store, and process data. Data repositories include relational databases such as MySQL [126], key-value stores such as Memcached [95], and others. The key challenge in these systems is scaling up to meet client demand, expressed in terms of throughput, and handling concurrent accesses to data correctly and consistently. We can meet this computational demand by using CPUs and GPUs while maintaining consistency. We pay specific attention to x86_64-based CPU architectures and NVIDIA's GPU architectures, although many fundamental parts of the architectures are consistent across CPUs and GPUs, respectively.

In the following sections, we detail the necessary background on the fundamentals of CPU and GPU architecture (Section 1.1), which will be required for determining ISA affinity. In Section 1.2, we will overview data repositories, their components, and essential considerations. Finally, in Section 1.3, we cover how we approach modern data repository design through the lens of ISA affinity of CPUs and GPUs.

1.1 CPU and GPU Architecture

CPUs must be designed to support operating systems and hypervisors through hardware features like ACPI and IOMMU. While GPUs also have hardware support for virtualization—allowing multiple processes to run simultaneously on a single GPU, their main design focus has been on accelerating computational tasks through an immense number of arithmetic logic units, floating point units, and specialized hardware units (e.g., NVIDIA's Tensor Cores).

In a heterogeneous system, the CPU is typically called the host, while a GPU or other co-processor is called a device. A process that uses a co-processor will begin on a CPU. The CPU will interact with the device through a driver, allocating appropriate resources, including memory. Once ready to compute, the CPU can queue the task, called a *kernel*, to be executed on the device with the machine code and arguments. This interaction will

typically occur through some interconnect, such as PCIe. A typical device will have at least one DMA engine to copy memory to and from the CPU. Once the memory is copied between the host and device, and the CPU has enqueued tasks, it can block till the device finishes the computation. This is our typical batch synchronous processing paradigm.

CUDA is the infrastructure that enables programming and execution on NVIDIA GPUs. The CUDA runtime and driver APIs provide stream objects on which to execute kernels. Streams are executed concurrently on the GPU and, depending on resource availability, can even enable parallel kernel execution.

For most GPUs, including NVIDIA's, memory copies must occur through memory mapped into pinned memory. The typical approach to performing a memory copy is to copy memory from the user to the kernel space and then perform the DMA. CUDA provides mechanisms to pin memory in user space to avoid this extra copy. CUDA also provides Unified Memory, which enables moving memory between the CPU and GPU through a page fault mechanism and oversubscribing GPU memory. In Chapter 2, we exploit both mechanisms to implement a high-performance key-value store.

While enterprise CPUs can have hundreds of cores, enterprise GPUs have ten-thousands of cores. Each of these cores is part of a larger unit in an NVIDIA GPU called a streaming multiprocessor, which can execute a maximum of 2,048 concurrent threads simultaneously. Modern GPUs like the NVIDIA H100 can have 132 streaming multiprocessors. Limiters such as register usage can reduce the number of concurrent threads, called *occupancy*, executing on these streaming multiprocessors; however, optimal programs should instead seek to utilize the entire compute throughput or memory bandwidth depending on what the limiter is. In most cases considered in this dissertation, the primary limiter is memory bandwidth.

When we program the GPU using CUDA, we partition our problem logically into computational blocks, then split further into threads. We illustrate this concept in Figure 1.1, as a GPU kernel operating on an image. We then assemble these blocks of computation into a grid of computation. In the figure, the computation grid is subdivided into 2 dimensions. A thread block processes each of these subdivisions in parallel. Each thread block of computation is mapped to a single streaming multiprocessor. Further dividing the thread block are threads, which we can again logically subdivide into two dimensions.

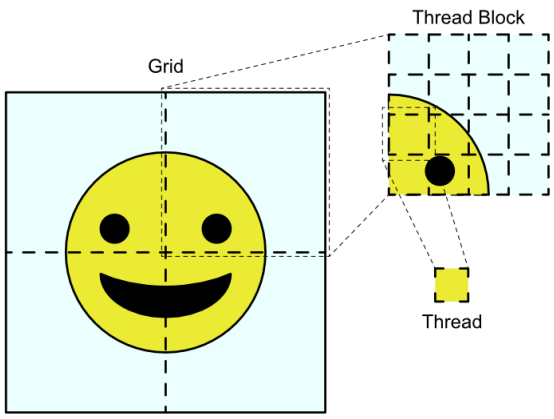


Figure 1.1: GPU Grid, Thread Block, and Thread Division when Performing Computation

Although threads are the smallest unit of processing within the CUDA programming model and GPU architecture, groups of 32 threads in NVIDIA's GPU architecture operate together in lockstep for optimal performance. This group of 32 threads is called a warp. Warps can perform intrinsics such as balloted voting across the warp for efficient synchronization. Furthermore, when a warp accesses a few cache lines, these accesses can be grouped or coalesced into fewer memory transactions. For example, the hardware will coalesce 4B accesses from a warp within a 128B cache line into a single cache access.

Each streaming multiprocessor possesses a chunk of scratchpad RAM called shared memory. This shared memory can be quickly accessed within a thread block. The GPUs possess synchronization primitives such as barriers and atomics to guarantee a consistent view of shared memory within the thread block.

In this dissertation, we consider problems primarily bottlenecked by memory operations. In many of our problems, the solution will require synchronization primitives, and consistency must be guaranteed. When determining whether the CPU or GPU is better suited to a problem, it is best to think through the following:

- The cost of moving memory between CPU and GPU may be non-trivial. The computation will have additional latency since at least inputs to a problem must be transferred between the CPU and GPU. Optimal solutions minimize this cost, and CUDA streams may be used to pipeline memory copies with kernel execution to achieve higher throughput. Memory management must be a primary consideration in designing and

determining how to solve a problem optimally with a GPU and CPU.

- The differences in memory bandwidth and latency on a GPU versus a CPU may make one or the other ideal. CPU memory bandwidth is significantly lower than the bandwidth on the GPU. This means that a highly parallelizable task bottlenecked by memory bandwidth should be able to achieve substantially higher throughput on a GPU. However, GPU memory latency is much higher than CPU memory latency. This means that CPU execution is faster for single operations if high throughput is not necessary.

- The potential for increased GPU contention can prevent scalability on the GPU. Since GPUs have at least an order of magnitude more threads executing simultaneously and contending on cache lines or primitives such as locks, algorithms that may scale on the order of 100s of threads can cease to scale at 1,000s or 10,000s of threads. This may make GPU performance suffer.

Throughout this dissertation, we rely on the above considerations in designing our systems, such as a CPU-GPU key-value store in Chapter 2 or a GPU-accelerated transactional data repository for emerging heterogeneous workloads in Chapter 3. Furthermore, when we approach sub-problems within data repository design, such as supporting high-performance transactional processing or data structures for these heterogeneous systems, we may either determine to only offload parts of our computation, such as in the transactional processing system OranguTAN in Chapter 4, or be agnostic and support both CPUs and GPUs, for example in CODS in Chapter 5.

To make the most use of the GPU and avoid necessarily determining that a CPU is better suited to a problem in our design of systems and algorithmic components, we specifically focus on the issue of increased GPU contention. To scale well on the GPU, we focus on using the relaxed memory model of the GPU [8] and its hierarchical nature [113]. Rather than simplifying our approach and aggressively fencing, we pay particular attention to what can be relaxed and still maintain correctness. This careful consideration allows us to design and evaluate our algorithmic approaches to modern data repository problems.

1.2 Data Repositories and Considerations

A data repository is a system that supports data storage, retrieval, and processing. Data exists within a spectrum of unstructured to structured. One type of unstructured data repository that we focus on heavily in this dissertation is a key-value store, where a key, such as a username, can map to a value, such as a password. Key-value stores do not contain any sense of relationship between data items.

Graph databases would be considered a more structured data repository, where data is related in a graph structure as edges and vertices. Relational databases are more structured, in which relational algebra describes relationships between data and how to operate. While this dissertation does not implement or design a graph or relational database, key-value stores are typically used under the hood of both types of data repositories, with higher-level software enforcing the relationships between data. This makes key-value stores a prime area to study regarding heterogeneous systems.

Users interact with data repositories through a variety of semantics. Two common semantics are operation-based APIs, for example, `GET`(key) to retrieve a value in a key-value data repository, and transactional-based APIs. Atomicity, consistency, and isolation are the highest priorities in a transactional-based system.

At a high level, atomicity refers to transactions happening in a single step. Consistency refers to how transactions observe one another's side effects. Isolation refers to how concurrent transactions interact with one another. Durability is also desirable, where a system can recover from a fault. While some databases guarantee atomicity, consistency, isolation, and durability— the well-known ACID properties, durability is not always guaranteed in data repositories. In this dissertation, we leave the question of durability as future work, instead focusing on atomicity, consistency, and isolation.

We consider both operation-based APIs and transactional semantics for data repositories. In KVCG (Chapter 2), the focus is typical key-value store APIs, including the common elementwise-linearizable `MULTIGET` API, where get operations are batched together into a single operation. This is done in practice in data repositories to achieve high throughput, and we find it is amenable to the GPU. In Chapter 3, we develop SnapKV, a transac-

tional key-value store that can execute various operations within its transactions, including deep-learning inference and relational algebra. SnapKV supports snapshot isolation.

We specifically focus on guaranteeing linearizability in our operation-based APIs and data structures. Both linearizability and snapshot isolation guarantee real-time order. With real-time order, reads and modifications must occur between the invocation and return of an operation or transaction. Furthermore, the result of the operation or transaction must be observable by other operations after the return. Real-time order is essential for modern workloads where it is desirable to have modifications visible as soon as an operation or transaction completes (e.g., systems that must handle fraud detection).

Linearizability also requires that the single operation appear atomic. The point at which the operation atomically occurs is called the linearization point. Operations that linearized before our operation cannot observe the results of the operation, while any operations that linearized after our operations must observe our side effects. Furthermore, with a linearizable data structure, we must guarantee the semantics of the data structure (e.g., an element can only be removed once).

In snapshot isolation, we linearize a transaction's reads at the beginning of the transaction, and any read must appear as though we have taken a snapshot of the data repository at this linearization point. Writes within our transaction will be linearized at the end of our transaction during commit. We further guarantee that two concurrent conflicting transactions cannot have an intersecting write set and both commit (i.e., two concurrent transactions cannot both write to key A). If this is the case, one of the conflicting transactions must be aborted and retried.

The stricter requirements of snapshot isolation and linearizability mean designers and implementers must be conscious of synchronization within any concurrent data structure or repository with these guarantees [77, 88, 167]. Any synchronization has the potential for slowdowns, which can impact performance in terms of latency or throughput. Designing optimal algorithms and data structures to avoid these bottlenecks becomes paramount when considering a heterogeneous system architecture.

We must carefully consider how we utilize the architecture and guarantee these desirable consistencies to support heterogeneous architectures as a first-class part of data repositories.

More specifically, we must understand our ISA affinity. We often find that our ISA affinity suggests that subparts of the system or algorithms we design are more suited to the GPU or CPU. With this in mind, we must synchronize between the CPU and GPU and guarantee a consistent view.

In KVCG, this requires routing operations to the CPU or GPU depending on the desirable performance characteristics. The CPU can achieve lower latency, and the GPU can execute with a higher throughput. We must do this consistently and handle the typical skewed workloads for data repositories.

SnapKV, on the other hand, opts for low-latency execution of data storage and retrieval on the CPU but offloads processing of the data (e.g., relational algebra) to the GPU. To support this, we utilize emerging trends in data structures, such as linearizable range queries. We introduce a new API called `SNAPSHOT`, which allows for efficient reads that can be offloaded to the GPU or used on the CPU for immediate processing, depending on the operation's ISA affinity.

We also focus on essential subcomponents of data repositories to enable the design of data repositories with heterogeneity as a first-class component. In Chapter 4, this entails understanding how to efficiently execute batches of transactions through understanding the semantics. Although not a core focus of this dissertation, we do this in the context of blockchain, where many transactions have well-defined semantics.

We further enable CPU and GPU-based data repositories by exploring data structures. We generalize a practical optimization called coalescence, whereby we access multiple data items simultaneously to reduce the number of memory transactions. We enable coalescence through a common framework across CPUs and GPUs and an emerging networking technology called remote direct memory access (RDMA). RDMA is used in data repositories for high-performance networking to enable distribution and fault tolerance. By generalizing coalescence, we demonstrate that siloed data structure research on CPUs, GPUs, and RDMA are, in fact, applicable across architectures and should be considered when designing heterogeneous data repositories. We also apply this framework in a practical data structure for linearizable range queries.

1.3 Roadmap

To approach data repository design through the lens of CPU and GPU ISA affinity, we begin with motivating systems work KVCG in Chapter 2 and SnapKV in Chapter 3. These works redesign primitives and approaches to better suit heterogeneous systems data repositories. KVCG utilizes both the CPU and GPU and routes requests between them to balance the high throughput GPU data structures and the low latency CPU data structures by learning the skew in key accesses and mapping highly accessed keys to the CPU and less frequently accessed keys to the GPU. SnapKV is a data repository that enables the acceleration of heterogeneous workloads, including hybrid transactional-analytical processing and online decision augmentation through heterogeneous processing on the GPU and a redesigned primitive called snapshot. Snapshot builds off of an ongoing research direction focusing on linearizable range queries for data structures. SnapKV provides real-time order guarantees through snapshot isolation.

Following these motivating works, we present techniques for improving the performance and accelerating essential parts of data repositories, including transactional execution and data structures. Transactional execution is essential to data repository performance, especially in data repositories replicating transactions in a distributed environment through state machine replication. Emerging data repositories support transactions with semantic guarantees, enabling acceleration through transaction commutativity and compensation. Blockchain is one such emerging data repository, supporting transactions with semantics related to currency exchange. Blockchain systems deal with data immutability and state machine replication across untrustworthy nodes. While dealing with these systems' modes of failure is outside the aims of this thesis, the acceleration of transactional processing, where transactions have specific semantics, is. In Chapter 4, we present OranguTAN, a semantic transactional execution system. We design OranguTAN in the context of Blockchain transactional processing, where currency is semantically sent and received between accounts. These semantics become the foundation for developing an algorithmic approach to executing these transactions with minimal overheads and mapping different parts of the overall system to CPU and GPU, depending on the ISA affinity.

Chapter 5 presents the CODS framework for generically designing and implementing coalesced data structures. Coalescence is a standard optimization in heterogeneous systems, which exploits architectural properties to reduce the number of memory transactions. This is the first work that we are aware of to generalize this optimization across CPUs, GPUs, and RDMA to design and study data structures.

Chapter 6 presents the lookup interlocked table, LIT. LIT is a data structure for linearizable range queries, similar to an interlocking hash table in how it chains and splits levels. Unlike hash tables, LIT is ordered and supports linearizable range queries. LIT's design is focused on being practical and supports coalesced optimizations through CODS.

Chapter 7 concludes the dissertation and summarizes the results and future research directions. We envision these future research directions to generalize approaches across architectures and domains within heterogeneous systems. Our work provides a starting point for balancing performance characteristics among architectures in KVCG (Chapter 2), introducing first-class heterogeneity in SnapKV (Chapter 3), exploiting parallelism, semantics, and ISA affinity in OranguTAN (Chapter 4), and designing hardware agnostic optimizations and data structures in CODS (Chapter 5) and LIT (Chapter 6).

Chapter 2

Designing Cooperative CPU-GPU Key-Value Stores

2.1 Introduction

Key-value stores play a fundamental role in many widely used applications that offer (possibly millions of) users the ability to access a shared state in a consistent way [65, 73, 34, 72, 117, 58, 98, 192]. Because of its flexible data model, and a simpler software architecture than traditional database management systems [172, 126], the adoption of key-value stores has rapidly grown over the last decade. Today, this class of data repositories must handle an extremely high volume of operations, which has motivated a large effort to improve performance [192, 133, 72, 110].

Looking deeper into the traditional key-value store operations (e.g., atomic `GET`, `PUT`, and `DELETE`) we find that they require few CPU cycles to execute. Despite their simplicity, the number of operations that need to be processed is large because of the high application demand.

The vital observation is that key-value store workloads suggest a lower affinity to CPUs than to accelerators, such as the Graphic Processing Unit (GPU). CPUs are designed to process fewer, but more complex, instructions than its GPU counterpart, which is evident when comparing the x86_64 ISA to PTX [47, 138]. On the other hand, the GPU architecture

has more, simpler, cores offering higher parallelism and can exploit a much higher memory bandwidth when compared to the CPUs as noted in [162].

Table 2.1: CPU and GPU Cost and Throughput of a Read-Only Uniform Workload (2022)

	Throughput	Cost
TBB Hashmap	66.2 Mops	$213 to $224
L-Slab Map	162 Mops	$229

GPUs are also more affordable hardware than large multicore architectures [135, 13], and nowadays they are also available in major cloud providers [12, 74, 120]. However, the high throughput offered by GPUs come at the cost of a high latency for individual tasks. In fact, well-known downsides such as traversing the external bus (e.g., PCIe) to transfer instructions and data between CPU and GPU, implementing efficient synchronization barriers, and launching kernels to execute on the GPU can stretch latency when offloaded tasks have short execution time, like key-value store operations.

We can observe both the affinity of accelerators to the key-value workload and the lower cost of GPUs empirically in Table 2.1. When running our own GPU hashmap against Intel's thread building blocks (TBB) hashmap [93] in a closed loop, we observe that the GPU can handle a higher throughput. The NVIDIA GTX 1660 Super GPU is able to perform this work at a similar price to the Intel i5 10600 [174, 94].

The traditional wisdom of Moore's law and Denard Scaling, which meant that CPU performance increases 2x every 18 months, no longer holds [60]. Furthermore, it is questionable whether increasing transistor density predicted by Moore's Law can hold much longer. However, GPU performance does increase exponentially as noted in Huang's Law [121]. Exploiting the affinity and low-cost of GPUs for this workload can guarantee that performance of GPU accelerated key-value stores on future hardware will be even better.

In this work, we present KVCG, a heterogeneous key-value store designed to leverage the relative merits of both CPUs and GPUs. KVCG aims to maximize the effectiveness of CPU resources while cooperating with the GPU. Our work builds on the observation that key-value store workloads are typically skewed, meaning a subset of keys is more popular than others [19, 45, 191]. This divides the key-space into frequently accessed (*hot*) and less frequently accessed (*cold*) keys. Hot keys require low latency and are accessed by

several application clients; cold keys can have a lower priority because they are not accessed with the same intensity. Previous works detailing real-world application behavior (e.g., at Facebook [133] and Twitter [191]) demonstrate that workloads have innate tendencies to access a subset of keys more frequently, and also that such a set of frequently accessed keys changes over time. These observations motivate the following design.

KVCG deploys three main components: the *Hot Cache*, a coherent software cache, accessed by CPU threads, to serve operations on hot keys; the *Canonical Store*, a data repository whose core component is a lock-based GPU hashmap (L-Slab) optimized for high throughput and able to store values of arbitrary size; and the *Router*, which relies on a model trained at runtime to classify incoming requests as hot or cold, and directs them to either the Hot Cache or the Canonical Store.

Operations that are not served by the Hot Cache on the CPU are batched and scheduled for execution on the GPU, where the Canonical Store operates. Effectively, the entire data repository resides on GPU accessible memory and keys are transitioned to/from the CPU based on the output of the model in the Router. In order to circumvent the size limits of GPU memory, KVCG's memory management exploits NVIDIA Unified Memory technology [161] to host both key-value pairs and metadata.

KVCG is implemented using CUDA 11.2 and evaluated using NVIDIA GTX 1080 and the YCSB benchmark [45] at different skews and mix of operation types (read/write). Our competitors are MegaKV [192], a key-value store specifically designed to exploit the GPU for providing high throughput; MICA [110], a high-performance CPU key-value store; as well as a CPU-only version and a GPU-only version of KVCG.

Overall, performance results show that KVCG is able to provide a latency for hot requests in the range 19us to 150us on typical read-heavy skewed workloads [191] (i.e., θ=0.5). The latency of cold requests also stays within the range 0.2ms to 2.2ms. Compared to MegaKV, KVCG achieves a 34x mean latency improvement with respect to the hot key's while retaining 1.1x throughput improvement with respect to all keys. Compared to MICA, KVCG achieves at least a 1.4x improvement in throughput.

When the workload changes during runtime, KVCG adapts its partitioning scheme between hot and cold keys by updating the model in the Router.

This paper makes the following contributions:

- We propose a novel design for effectively deploying a key-value store across both the CPU and GPU to accelerate skewed workloads.
- We demonstrate that isolating hot keys to the CPU and processing requests targeting cold keys on the GPU improves throughput while dramatically lowering latency for operations on hot keys.
- We present L-Slab, an extension to the high-performance GPU-based hashmap, Slab-Hash [18], that supports arbitrarily sized values on the GPU.

2.2 Related Work

Both academia and industry have proposed many key-value stores in the last decade. Examples include Memcached [65], LevelDB [73], RocksDB [34], EvenDB [72], Oak [117], FARM [58], FaSST [98].

Mica [110] is a related key-value store proposed in academia that attempts to handle key-value operations and associated skew by statically partitioning keys to CPU cores. As requests come in, keys are hashed to cores, which provides a mechanism to load balance requests and limit the affects of skew. Mica can operate in an exclusive read-exclusive write mode where accessed are exclusively mapped to their partition/core, a concurrent read-exclusive write mode where reads may be mapped to any core and read a different partitions map, and a traditional concurrent read and write mode. Unlike Mica, KVCG focuses on load balancing across heterogeneous hardware and does so by modeling the distribution of requests rather than load balance through hashing.

With the advent of key-value stores, data structures with map semantics have become increasingly important. Skip lists (e.g. NUMASK [51], a GPU warp-cooperative skiplist [125], No Hot Spot [50]) have become highly popular ordered maps in the literature due to their simplicity and applicability. Hashmaps have also been important data structures in research (e.g. Michael's hashmap [118], interlocking hash table [96]). Most similar to our GPU hashmap we propose in Section 2.5.1, skip vector [156] utilizes blocks of key-value pairs, which they call vectors and we call slabs. Unlike skip vector, our GPU hashmap is

unordered and designed specifically for high-performance on the GPU.

Since KVCG focuses on cooperation between the CPU and GPU to serve requests, in the rest of the related work we focus on previous effort in executing atomic operations on heterogeneous devices.

In recent years, synchronization on the GPU has gained substantial attention. Transactional memory [31, 69, 37, 187, 42, 35], lock-based synchronization [189, 108, 192] and lock-free approaches [186, 122] have all been explored as viable options for designing concurrent applications for the GPU. Furthermore, evidence that fine-grained synchronization can benefit GPU programs provides a foundation for the continued efforts along this line of research [129, 108]. KVCG follows suit by allowing write operations to perform concurrently with read operations on the CPU and GPU.

MegaKV [192] is a key-value store that handles operations on the GPU and uses a static dispatching policy to provide predictable performance. It is similar to KVCG in its aim to accelerate key-value stores. It utilizes a concurrent cuckoo hashmap and a protocol that deterministically schedules GPU kernels to execute batches of requests. The cuckoo hashmap is similar to our L-Slab hashmap in its use of warp-cooperative processing. The cuckoo hashmap differs in its lock-freedom and ability to index only 4B keys and 4B values. KVCG is able to utilize the oversubscription of Unified Memory to provide indexing that can extend beyond the limits of GPU memory, while MegaKV must evict items when GPU memory fills up. Unlike MegaKV, KVCG is able to utilize the CPU for performing low-latency operations on hot keys.

Because it is constrained to atomic operations for key-value pairs, MegaKV only stores 4B hashes to 4B pointers on the GPU, which forces MegaKV to resolve collisions on the CPU in post-processing. Instead, KVCG reduces overhead by utilizing a lock-based approach and allowing for the map to index any size key or value on the GPU.

Closer related works to KVCG are SlabHash [18], a hashmap based on a GPU data structure called a slab list, and HCC [30], a hybrid cache coherent hashmap. SlabHash is extensively described in Section 2.5 since KVCG includes an improved version of SlabHash, named L-Slab, to process operations on the GPU. Instead, here we focus on HCC. HCC uses the CPU to perform write operations in a non-blocking fashion while `GET` operations

are scheduled on the GPUs. Similar to KVCG, HCC also uses Unified Memory but differs in that it offloads `PUT` operations to the CPU. Therefore, for HCC, the GPU workload is read-only, whereas KVCG processes both read and write workloads concurrently on the GPU. HCC is specifically optimized to take advantage of IBM's Power9 architecture with NVLINK; KVCG targets off-the-shelf hardware.

2.3 System Overview

KVCG implements the following widely used atomic linearizable APIs: `GET`, `PUT`, and `DELETE`. In KVCG the return value of a `GET` is the data associated with the provided key. The `PUT` and `DELETE` APIs return the value of the provided key before the update takes effect, if any existed.

KVCG is designed to take advantage of the high throughput of the GPU and the low latency of the CPU. At a high level KVCG includes three main components.

- The *Canonical Store*, which holds all the key-value pairs in the data repository. This store is has values placed in CPU memory when values are over 8B, with an index accessible by the GPU. The Canonical Store also includes a concurrency control to let GPU threads process atomic read and write operations on the key-value pairs. KVCG avoids limiting the total size of the index to the memory resident on the GPU by relying on NVIDIA Unified Memory (UM) [161]. UM enables a unified view of the address space and on demand paging across both the CPU and GPU since CUDA 6 [81]. Through a combination of UM and explicit memory management, KVCG can utilize more memory than is available on the GPU while avoiding excessive paging overheads [109].
- The *Hot Cache*, which serves operations on *hot* (i.e., frequently accessed) keys to ensure low latency for popular requests. The Hot Cache is coherent, meaning it caches the latest value of current hot keys. This cache allows KVCG to effectively exploit the available CPU threads (significantly outnumbered by GPU threads) to perform low-latency operations without the interference of requests targeting *cold* (i.e., less frequently accessed) keys, which would otherwise saturate CPU threads.
- The *Router*, which includes a model to determine whether or not a key should be destined

for the Hot Cache or the Canonical Store. Importantly, a wrong prediction by this model does not impact operations' correctness, it only affect their performance. The model is continuously trained with incoming client requests so that KVCG can dynamically change the composition of cold and hot storage to respond to application workload changes over time.

To accommodate the needs of modern key-value stores, KVCG allows clients to issue a batch of requests as opposed to individual operations. This mechanism is a generalization of batched operations, like the multi-Get API supported in Memcached [65], a well-known key-value store. Our expectation is that KVCG is paired with a middleware that aggregates end-user requests into batches that are then submitted to the store. We call these batches *Client Request Batches* (CRB).

In the following sections, we first describe the lifetime of a CRB then follow up with a discussion of the design of each of the components that make up KVCG.

2.4 Execution Workflow

This section follows the journey of a CRB as its requests are processed by KVCG. Each entry in the CRB consists of the request type, the key-value pair, and a response field. The latter is used to notify the client that the request has completed successfully. A response either informs the client that the operation completed or that the operations should be retried. As we show later, a retry is required in two cases. First, when the GPU cannot accept new requests to avoid overrunning memory. Second, when updates are blocked during a model change.

Figure 2.1 illustrates the software architecture of KVCG and embeds the sequence of steps performed by client requests to accomplish their execution.

① The first stage of processing is partitioning all requests in a CRB into hot requests or cold requests based on the current knowledge of the Router. Note that this partitioning scheme can change over time as a consequence of an update to the Router's model (more details in Section 2.7).

② After partitioning, the cold requests are batched in a GPU Request Batch (GRB)

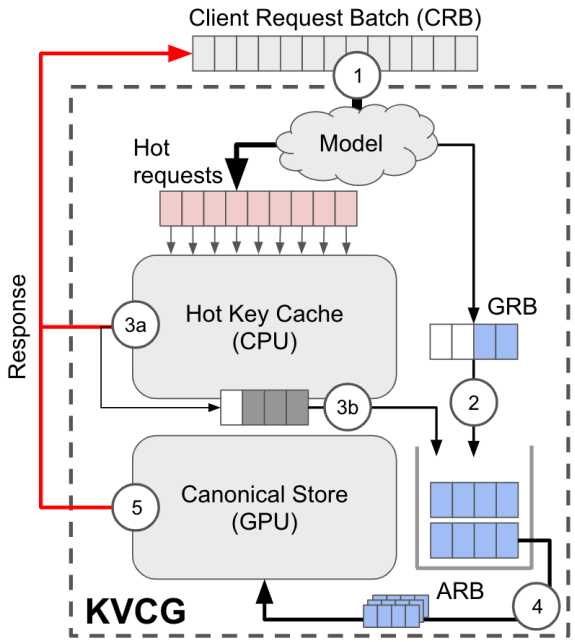


Figure 2.1: KVCG architecture and request workflow.

and enqueued for execution on the Canonical Store through a GRB queue.

(3a) Requests that were not sent to the Canonical Store are concurrently executed on the Hot Cache by a pool of dedicated threads. If a request can be serviced by the Hot Cache, meaning there was a hit, then the client is notified immediately. Otherwise, misses are batched for execution on the GPU where the Canonical Store resides, similar to the cold requests. The term GRB also applies to these batches.

(3b) Requests that missed the Hot Cache are enqueued on the GRB queue for processing on the Canonical Store.

(4) As GRBs are enqueued, dedicated threads are responsible for dequeuing them from the head of the queue and creating *Aggregated Request Batches* (ARBs). ARBs are necessary to fully leverage the massive parallelism of the GPU.

(5) Finally, ARBs are fed to the GPU. Requests are executed in parallel with the appropriate support for mutual exclusion to ensure atomicity of operations. At this point and if needed (see Section 2.6), the Hot Cache is updated to avoid subsequent misses. The return values of operations is transferred through the value field of the corresponding request in the ARB to clients.

According to the just described execution flow, the CPU and GPU concurrently execute

requests. Consistency is guaranteed because effectively the key-value store is partitioned at any given moment between the Hot Cache and the Canonical Store. The only potentially dangerous scenario for correctness is when the model in the Router is updated. We will detail how we handle that in Section 2.7.

2.5 The Canonical Store

The Canonical Store supports `GET`, `PUT`, and `DELETE` operations on a given key. The core component that guarantees the correct and effective execution of these operations is a GPU lock-based hashmap, we name *L-Slab*.

L-Slab relies on readers-writer locks [129, 8] to enforce ordering between conflicting operations, and warp-cooperation [18] for increasing performance of the lookup functionality (Section 2.5.1). L-Slab uses 8-byte keys and values, meanwhile exploiting Unified Memory to support a total capacity exceeding that of GPU memory alone. When variable sized values are required, pointers to non-adjacent memory can be used (see Section 2.5.1).

As described in Section 2.4, a GRB queue is used to accumulate requests that are either directly passed to the GPU processing pipeline via the Router, or requests that missed in the Hot Cache. As GRBs are received, they are formed into the larger ARBs for dispatch to the GPU. As long as there are GRBs in the queue, a thread will continue to fill its ARB until there are twice as many requests as the number of threads available on the GPU or no more GRBs are available. To avoid blocking in the latter case, we deploy a dequeue budget that limits the number of times a queue is checked.

ARBs are aggregated batches of requests that contain the target key, the key hash, an associated value, and the request type (i.e., `GET`, `PUT`, `DELETE`). Since the ARB is accessed by the GPU, it is first provisioned in pinned host memory, then copied to the GPU with a call to `cudaMemcpy`. Upon completing execution on the GPU, the results of each operation are then copied back to the ARB in host memory before returning to the client. Explicit memory management allows for more efficient ARB processing because all request data and metadata resides on the GPU when the kernel is launched. We decided against allocating ARBs using Unified Memory because it incurs the additional cost associated with paging

memory to the the GPU, which negatively impacts performance when interleaved with request processing.

ARB dispatch is handled by *orchestrator threads*, which handle ARB creation, GPU kernel launch and client response. In our implementation, we associate each orchestrator thread with a CUDA stream [136] to pipeline execution. The above choice is critical to minimize the impact of the overhead of offloading work to the GPU, therefore retaining low latency while still enjoying the high throughput of the GPU.

2.5.1 Lock-Based Warp-Cooperative Slab Hashmap

L-Slab is a lock-based warp-cooperative hashmap data structure and builds upon the GPU-based SlabHash hashmap [18]. As opposed to conventional hashmap designs, SlabHash is appropriate for GPU because each bucket stores a linked list of *slabs*, and each slab holds 31 key-value pairs and a pointer to the next slab in the bucket. This design matches the GPU execution structure where an entire warp, meaning a set of 32 GPU-threads, can be assigned to work on a slab. In order to atomically modify key-value pairs in the hashmap, SlabHash uses Compare-And-Swap (CAS) operations. Because of that, it can only accept key-value pairs of size 8 bytes total (e.g., 4-byte for the key and 4-byte for the value), which is the hardware limitation to guarantee correctness of the CUDA CAS operations. Accommodating larger size of key-value pairs in SlabHash would require a secondary index to be queried using the original 8B key-value pair.

L-Slab inherits SlabHash's use of warp-cooperation because *i)* it provides high performance due to favouring coalesced accesses to the GPU memory [83]; and *ii)* GPU threads within a warp can use fast hardware instructions (e.g., `__shfl_sync`, `__ballot_sync`) to coordinate their activities and guarantee correct concurrent accesses over shared data. Unlike the lock-free design of SlabHash, L-Slab deploys a reader-writer spin-lock to protect each bucket. This modification makes KVCG more practical because it can efficiently work with any size of key-value pairs, as we will show later in Section 2.5.1.

L-Slab serves requests of all types with the following general pattern. At each kernel launch, the ARB is first partitioned among warps. We oversubscribe the streaming multiprocessors (SMs) in the GPU by launching as many blocks as twice the number of SMs,

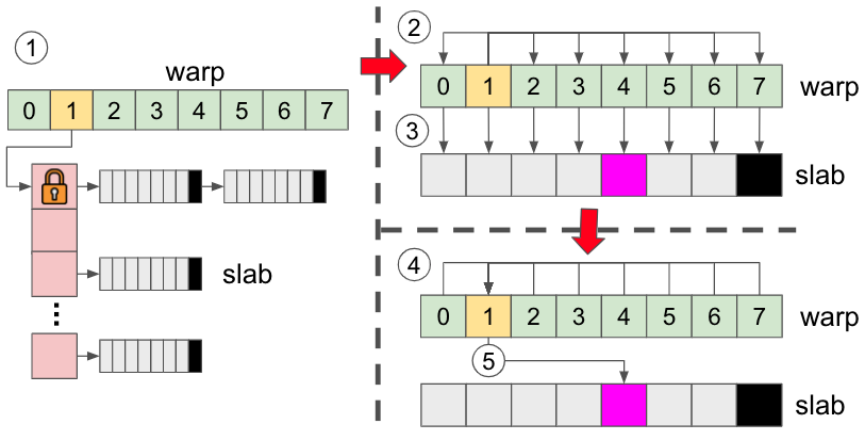


Figure 2.2: Execution of a `GET` operation on L-Slab.

which offsets the latency involved with launching a kernel and transferring the ARB to the GPU. Each block consists of 16 warps and we assign each thread a single operation to perform.

During execution, the warp works in unison to complete all requests that its threads are assigned, one request at a time. For each request, a single thread within the warp orchestrates the operation. Per-bucket locks allow threads in a warp to safely cooperate despite conflicting concurrent operations.

Figure 2.2 demonstrates this execution for a `GET` operation, with a simplified warp consisting of 8 threads. ① Initially, the *leader thread* (in this case Thread 1) locks the bucket corresponding to its request. ② Then, it broadcasts the target key of its request to the entire warp. ③ Together, all threads in the warp check their corresponding entry in the first slab for the broadcast key. ④ Next, all threads in the warp report back to the leader whether their entry matched the key. ⑤ If the key is found, then the leader performs the `GET` and releases the lock. If the key is not found, and there is a pointer to another slab, then the leader broadcasts the address of the next slab and the search continues. Finally, this entire process is repeated for every request designated to the warp, with each warp thread acting as the leader for its assigned requests. We will discuss `PUT` and `DELETE` operations in more detail later, but we first explain the layout of slabs.

Each slab in our design is composed of two distinct arrays that are adjacent in memory. Considering the case of keys and values of 8B each, a slab is constructed as an array of

32 8-byte entries, 31 are reserved for keys and the last for the pointer to the next slab, followed by an array of 31 values. This layout is motivated by the following intuition. For any request, a value is accessed (i.e., read or written) only after the slab's keys have been analyzed and matched with the key that was broadcast by the leader of the current operation. Packing keys together allows accesses by the warp to be coalesced, which is a well-known optimization to improve performance on the GPU [83]. Once found, the location of the value corresponding to the key can also be easily determined, since each entry in the slab is a fixed size. We describe how we leverage these entries to store arbitrary sized keys and values later in this subsection.

We now detail how each operation on L-Slab is handled. As described above, the warp cooperatively indexes to the bucket corresponding to the hash. The first thread in the warp acquires the lock on the bucket. Then, the warp proceeds to search for the key or an empty key if the key cannot be found for a `PUT`. One of the following then occurs:

- `GET`. The leader reads the value associated with the key and writes it to the ARB.
- `PUT`. The leader writes the key and value to the slab, and writes back the previous value, if any, to the ARB.
- `DELETE`. The leader sets the matching key, if found, to the empty key and writes back the previous value to the ARB.

After the operation finishes, the lock is released.

Handling large size key-value pairs

As mentioned earlier, L-Slab can support keys and values of arbitrary sizes. In the above description we assumed that keys and values have both a fixed size of 8 bytes (if less, they are padded to 8 bytes). If larger sizes are desired, the keys and values in the map are pointers to immutable memory blocks that can be unrestrictedly allocated either in UM, GPU, or main memory.

The only additional overhead of using values larger than 8 bytes, which therefore cannot be embedded into the internal memory layout of L-Slab, is the memory copy performed before delivering the return value of an operation back to the application on the CPU.

2.5.2 Memory Management

As said earlier, the Canonical Store leverages Unified Memory (UM) for over-subscription, which allows more memory than the one physically available on the GPU to be allocated. Utilizing UM introduces advantages and flexibility through a hardware-implemented on-demand page fault mechanism that allows the CPU and GPU to cooperate over a coherent shared memory. The downside of this increased flexibility is the possible performance penalty due to repeated page faults that let memory pages continuously migrate between devices [109]. Another issue of UM is that memory cannot be dynamically allocated by GPU threads. This capability is critical for KVCG since slabs in the L-Slab hashmap should be allocated/deallocated at runtime to accommodate update operations on the GPU without involvement of the CPU.

By taking into account the above limitations, KVCG leverages UM as follows. In order to allow for dynamic memory allocation on the GPU, regions of memory are pre-allocated in UM. At runtime, GPU threads can request chunks of the pre-allocated memory. If the requested amount exceeds the total memory previously accounted for, then the CPU reclaims the kernel and expands the pre-allocated region.

In addition to that, UM is used by KVCG only for allocating the actual L-Slab hashmap structure (i.e., the index) and the key-value pairs. ARBs are not allocated in UM because their size is often smaller than the granularity at which UM transfers memory between CPU and GPU. Since the performance of operations involving the ARB is critical for achieving low-latency GPU computation, we prefer to explicitly move ARBs from CPU to GPU and vice-versa.

2.6 The Hot Cache

The Hot Cache wraps a high-performance concurrent data structure to hold some key-value pairs of KVCG. The purpose is to provide low-latency operations for a subset of keys that are classified as *hot* (i.e., frequently accessed) by the Router. It should be noted that the integration of this component into KVCG is made in a way the Hot Cache design and implementation can be independently replaced and optimized with respect to KVCG.

The Hot Cache is implemented as a readers-writer lock-based hashmap, where each bucket consists of a linked list of nodes containing 8 elements each. The hashmap is pre-allocated based on an initial memory budget, but can expand beyond that in the case of conflicts. However, given that each node consists of 8 elements, we expect this to be rare.

After locking the bucket corresponding to the target key, each operation performs its necessary work then releases the lock. Operations on keys that are present in the cache simply update, delete or return the existing value. When there is no entry corresponding to the key, then some additional steps are required. In more detail:

- `PUT`. If the entry already exists, then it simply updates the value. A `PUT` operation on a non-resident key reserves an unused entry – allocating a new node as necessary – and populates it with the value contained in the request.
- `DELETE`. For existing keys, the entry is simply marked as deleted. For non-cached keys, a `DELETE` will reserve a new entry – again allocating nodes as needed – and mark it as deleted to serve as a tombstone for future operations. A subsequent `GET` can therefore be serviced immediately without querying the Canonical Store. Similarly, subsequent `PUT` operations need not to allocate memory for the new key, but can operate directly on the existing entry.
- `GET`. If the bucket contains an entry corresponding to a key, and the value is not logically deleted, its value is returned. When a miss occurs (i.e., there is no entry for the key) the request is added to a GRB to be executed on the GPU. A special flag indicates that this request originated from a miss and instructs the GPU thread responding to populate the cache after the operation completes.

Recall that the key space is partitioned between the Hot Cache and the Canonical Store, hence requests remain consistent through the underlying concurrency control of each storage component. However, when the model changes, all `DELETE` and `PUT` operations made on the Hot Cache must be propagated to the Canonical Store at the moment the model changes. Hence, in addition to generating and updating entries in the Hot Cache, update operations (i.e., `PUT` and `DELETE`) should be logged. This log is implemented as a secondary hashmap with a one-to-one mapping between it and the Hot Cache. It is stored in contiguous memory to ease scan operations and capture the most recent updates made to keys on the Hot Cache

that must be propagated to the Canonical Store. The next subsection describes this process.

2.6.1 Replaying Logged Operations

The goal of the Hot Cache is to make frequently accessed keys available to application requests so that they can be executed with low latency. Our system is designed to do so while adapting to changing workloads by dynamically updating the Router's model in response to new patterns. During this process we must ensure that operations maintain correctness. With keys whose hot-cold assignment does not change, there is no risk of inconsistency since each component guarantees linearizability. For keys that were originally cold but become hot, the cache miss policy will provide the most up-to-date version from the Canonical Store. However, if a key was previously hot but now cold, any future request routed to the GPU must observe the most recent value.

The moment the model is updated marks the start of a new epoch for requests. Requests that had previously been hot may now be cold and therefore destined for the GPU. As such, it is crucial to maintain the consistency as this change is made. Before new requests can be served, any update to the Hot Cache must be reflected in the Canonical Store. As mentioned, we maintain a log of updates performed during the previous epoch. These logged requests are batched and enqueued on the GRB queue. Once the log is replayed on the Canonical Store, new update requests can be served, since any operations going to the Canonical Store will be ordered after the replayed log. Note that it is sufficient to only replay the last epoch as requests of prior epochs will already be present on the Canonical Store. In other words there is only a single epoch of requests outstanding on the Canonical Store.

2.6.2 Cleaning the Hot Cache

Another important part of ensuring consistency is the removal of cache entries for keys that are no longer hot. Under our assumptions, the size of the entire data repository can fit in main memory. A consequence of this is that evictions from the Hot Cache are not performed to make room on the cache, as traditionally done [110, 192], but rather to remove keys that are no longer hot. This work is performed by a helper thread every time the model in the

Router changes. When that happens, the thread iterates over the Hot Cache removing all entries that are cold according to the new updated model.

To guarantee correctness (i.e., linearizability) of operations, the helper thread must complete its eviction before any subsequent model changes. The above requirement combats the case in which some model, $m1$, specifies a given key as hot; its successor, $m2$, classifies the key as cold; then a third model, $m3$, again labels the key as hot. In this scenario, if the key is not evicted from the cache when $m2$ is active, then an operation directed to the Hot Cache by $m3$ may observe a value written when $m1$ routed requests. All keys that are no longer hot are removed from the Hot Cache upon each model transition to ensure no stale values are read.

It is important to note that KVCG exploits the massive parallelism of the Canonical Store when replaying the logged operations. Accordingly, we envision a rapid transitioning process (e.g., < 500 ms in our experiments).

2.7 The Router

KVCG relies on the Router to decide which keys are *hot* and which are *cold* at runtime, and route them to the appropriate store for execution. A machine learning based classifier, denoted in this document as the model, is integrated into the Router and its purpose is to accept a key and the hash of that key as input and simply return a classification of that key as hot or cold. KVCG then adapts to reflect the classification made by the Router. Thanks to its design, other models can also be used in the Router to tailor it to the application's needs.

While running KVCG, a shadow model is trained by sampling incoming requests. Periodically, the shadow model replaces the current model with the goal of continuously adapting the partitioning of the store between Hot Cache and Canonical Store in order to meet workload changes.

Replacing the current instance of the model is done without impacting the correctness of ongoing operations. In order to do that, a lock is held for the entire duration of the process to prevent triggering another model change. Once the lock is acquired, all incoming update

operations must retry on the Hot Cache while ongoing operations and new `GET` operations complete on the Hot Cache. After that, `PUT` and `DELETE` operations on the Hot Cache are replayed on the Canonical Store utilizing the process described in Section 2.6.1.

After that, the model can be updated and the Hot Cache resume serving all operations. Evictions of no-longer-hot keys should now occur on the Hot Cache; this is done using the procedure described in Section 2.6.2. After the eviction is complete, the lock is released.

During the change of the model, the Canonical Store and Hot Cache can execute previously enqueued operations. This is because any request previously enqueued for execution on the Canonical Store will be able to access the most up-to-date values by relying on the step (described previously in Section 2.6.2) that replays the latest Hot Cache operations on the Canonical Store. After the Hot Cache resumes execution, all operations previously batched and still to be executed on the Canonical Store are effectively concurrent with operations to be executed on the Hot Cache. Correctness is still guaranteed in this case, as shown below.

Let us consider a key k previously classified as cold and currently hot according to the new model. Let us also consider an operation o_i on k that was batched for execution on the Canonical Store before the change of the model started. If a new conflicting operation o_j is issued after the model is changed, then it will be routed to the Hot Cache. At this moment, o_i and o_j are conflicting and operating concurrently on the two stores of KVCG. In this case, linearizability is maintained since these operations are concurrent thus no order must be enforced between them. Anyway, due to the installation of the new model, all new requests will then be routed to the Hot Cache and therefore any possible update of the value of k in the Hot Cache will be the one finally read by future operations. Eventually, future evictions will reflect the most recent value of k in the Canonical Store.

In our implementation, we use a histogram trained on the incoming client requests at runtime. We set a threshold where any bucket of keys accessed at a proportion above the threshold is classified as hot (e.g. a threshold of 0.1 means that any bucket where the proportion of total accesses is greater than 0.1 is classified as hot). We find experimentally that a well-trained model alleviates contention on the Canonical Store and on the Hot Cache, especially in the presence of skewed workloads (see Section 2.8.4 in the evaluation

study).

2.8 Evaluation

We evaluate the design of KVCG by comparing performance with our primary GPU-based competitor MegaKV [192], as well as a high-performance CPU-only concurrent hashmap, MICA [110]. Additionally, we include two special configurations of KVCG that correspond to a GPU-only version (i.e., just the Canonical Store), we denote as *KVG*, and a CPU-only version (i.e., just the Hot Cache), which we name *KVC*.

The testbed consists of two Intel Xeon Platinum 8160 CPUs, totaling 48 available cores, with hyperthreading disabled, and a GeForce GTX 1080 GPU with 2560 CUDA cores across 20 SMs. For each competitor, we tune the number of CPU threads and GPU streams to maximize their respective performance. All systems, except MICA, use a batch size of 512. MICA leverages all available CPU threads and extends the batch size to 4096 requests to guarantee high utilization; KVCG utilizes only a single NUMA zone (24 cores) for the Hot Cache to reduce hardware contention. Because of its pipelined architecture, we empirically verified that MegaKV performs best with 10 CUDA streams. In addition to the 24 cores dedicated to the Hot Cache, KVCG also allocates 10 threads to handle GPU streams on the Canonical Store.

Unless otherwise stated, the key-range of each competitor is fixed to 1 billion keys to reflect large stores, the hash table is prefilled with 10 million key-value pairs. We report both latency and throughput results averaged across 512 million requests executed in a closed loop. Update operations are performed such that the final load factor is close to the starting load factor. All experiments measure performance under a workload following a Zipfian distribution provided by the YCSB benchmark [45]. To emulate application workloads with varying skew, we report results at different values of theta (θ). Finally, all competitors use the same hash function: $H(x) = x\%s$, where s is the size of the data repository.

KVCG is implemented from the ground up in C++ and the GPU kernels are written using CUDA 11.2. We also reimplement our primary competitor (i.e., MegaKV [192], avail-

able at [193]) in the same environment. We replace the underlying GPU hash table (i.e., `libgpuhash`) with the same SlabHash hash table that is used for KVCG's Canonical Store. Table 2.2 demonstrates that even for a single-warp block, this replacement improves `GET` and `PUT` performance by about 2x, albeit `DELETE` operations are slower because MegaKV's `libgpuhash` leverages a lock-free logical deletion. Under the read-intensive workloads evaluated, we believe a SlabHash-based MegaKV to be a more suitable competitor.

Table 2.2: SlabHash versus MegaKV's `libgpuhash`

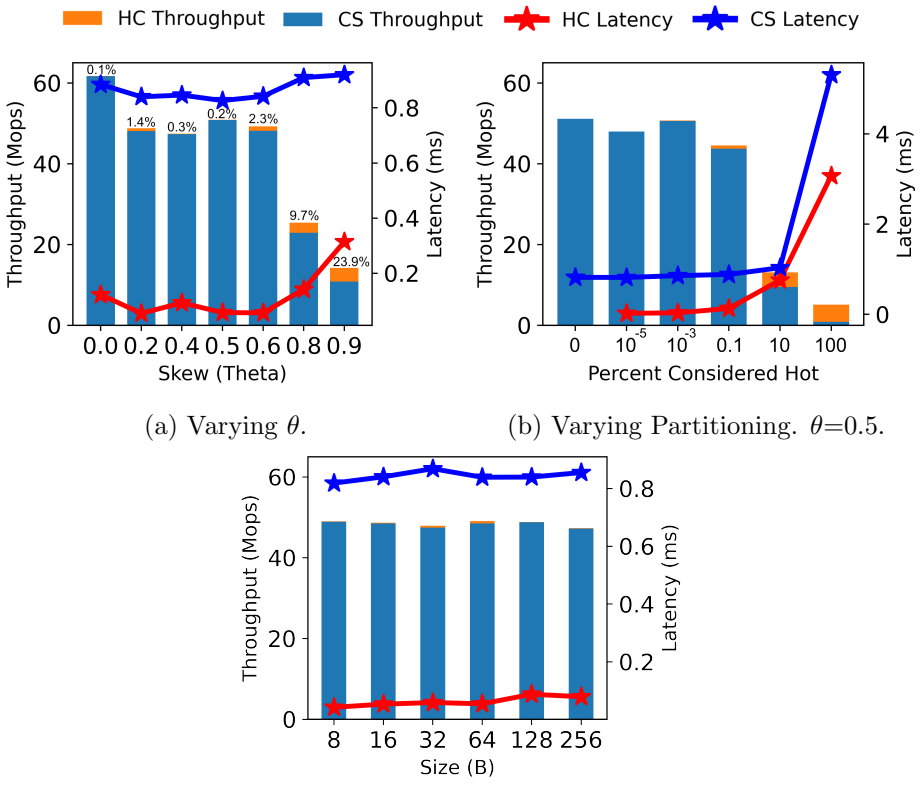
Operation Throughput	SlabHash	`libgpuhash`
PUT (MOPS)	61.0	27.3
DELETE (MOPS)	120.4	233.5
GET (MOPS)	269.7	160.0

In all of our experiments, we implement KVCG's model as an approximation of the workload to capture the incoming requests without matching them perfectly. To do so, we use a histogram model that splits the key space into 10,000 bins. During training, a bin accessed at a proportion above a predefined threshold is classified as hot. At inference, if a request's target key falls within a hot bin, the request is routed to the Hot Cache. Note that when calculating performance metrics, a miss in the Hot Cache counts towards the Canonical Store since it is the first time accessing this missed key after the model changed and the request is loading it from the GPU. Misses are expected to occur infrequently if the model accurately reflects the application workload.

The rest of the evaluation section is organized as follows. First, we explore KVCG in isolation with 8-byte keys and values in Section 2.8.1. Then, in Sections 2.8.2 and 2.8.3, we explore how KVCG compares to competitors. Later, in Section 2.8.4, we explore how the model impacts the performance of KVCG. In the legends of the included plots, we use `HC` to refer to the Hot Cache and `CS` to refer to the Canonical Store.

2.8.1 Analysis of KVCG performance

We begin our discussion with an experiment to demonstrate the performance of KVCG for 8-byte values across various workloads. Figure 2.3a shows how our system responds to an increase in skew. For this experiment, we use the aforementioned histogram model from



(a) Varying θ.

(b) Varying Partitioning. θ=0.5.

(c) Varying Value Size. θ=0.5.

Figure 2.3: Throughput and mean latency of KVCG. Reads are 95%; Writes are 5%. Skew is noted as θ.

Section 2.7 with a threshold of $1.3 * 10^{-5}$ to approximate the incoming workload. For each skew, we retrain the model on 5 million requests.

In any key-value storage system, we expect that the overall throughput decreases as the skew becomes greater because of higher contention on the internal data store. KVCG is able to adapt to this change by offloading hot requests to the Hot Cache, which can be observed in the steady increase in the Hot Cache's overall proportion of the achieved throughput. Although latency of requests to hot keys increases with the skew, it remains below 0.15ms until a skew of 0.9; after that, it only increases up to 0.32ms. Because of the massive parallelism offered by the Canonical Store, and the general lack of contention, we observe the highest overall throughput at the lowest skew. However, this comes at the cost of higher latency due to more processing on the GPU. While routing requests, each classification takes 315 ns on average.

Next, in Figure 2.3b we measure throughput and mean latency of the system when fixing the skew to θ=0.5 and varying the percentage of keys that are classified as hot by the Router. We choose 0.5 ($\alpha = 2$) since this is the highest theta reported by Twitter in their read heavy workloads [191]. The model for this experiment uses prior knowledge, corresponding to the Zipfian distribution used to generate the workload, to classify a fixed proportion of the most popular keys as hot. Because of the skew, a small fraction of hot keys may result in a larger proportion of overall requests. In other words, when the Router only considers 0.1% of the key space as hot, the Hot Cache serves 1.83% of the overall requests. An interesting trend emerges when we consider a scenario in which all operations are routed to the GPU only. In this case, throughput remains relatively high, but at the cost of an average latency of 0.82ms.

It is of note in Figure 2.3b that KVCG performs best when few operations are fulfilled by the Hot Cache. When many requests are routed to the Hot Cache, the GPU utilization is low, and the Canonical Store is unable to provide high throughput. With few requests routed to the Hot Cache, however, few requests get low latency. When running KVCG, we look to find the trade-off between throughput and latency. By routing the hottest 0.1% of requests to the Hot Cache, we provide much lower latency (i.e., 131.7us) for those requests while retaining similar latency and throughput on the GPU.

Up to 0.1% of the keyspace being considered hot, the throughput is greater than 40 Mops. Between 0.1% and 10%, there is a performance drop because the Canonical Store is not receiving enough requests. To the extreme, when all requests are served by the Hot Cache, the throughput is low and the latency is high because contention prevents the Hot Cache from performing well. We do notice some of the requests where 100% of the requests are routed to the Hot Cache are executed by the GPU. This is because compulsory-misses, misses that initially occur because the key is not present in the Hot Cache, occur and are counted as GPU execution.

These trends help to highlight both the importance of the model and the impact of contention on the Canonical Store. If the model classifies too many operations as hot, then it can have a detrimental impact on performance. While the Hot Cache provides superior latency over the Canonical Store, the massive parallelism provided by the GPU is important

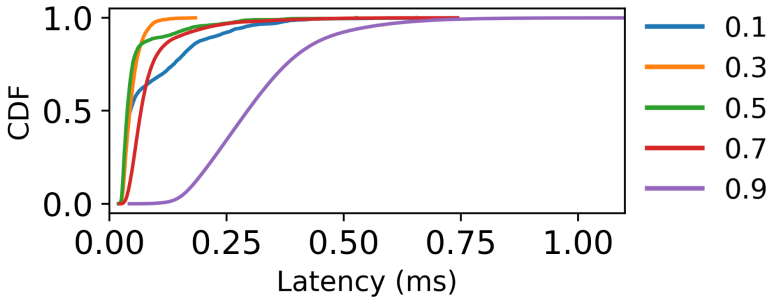


Figure 2.4: CDF of Hot Cache. 95% read ratio. 8B values.

to support higher throughput, especially when operations are non-conflicting. In doing so, KVCG can achieve good performance relative to the respective needs of the operations.

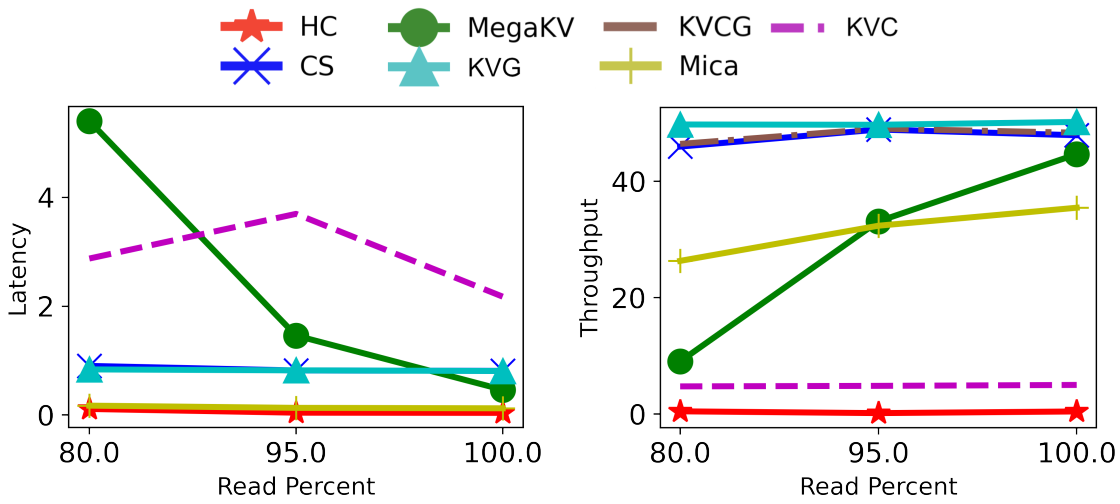
When considering the impact of value size on performance in Figure 2.3c, we find that the size of the value up to 256B minimally impacts KVCG performance. The same trend emerged for all other competitors, which also rely on indirection to handle variable-sized values.

We also explore the cumulative distribution function, CDF, of the latency for the Hot Cache in Figure 2.4 and find that while serving a lower proportion of requests (a lower theta), the latency is lower at most points. As the theta increases, a higher proportion of requests are served on the Hot Cache and there is contention on the keys. This creates a greater tail latency.

2.8.2 Varying Read/Write Ratio

We evaluate KVCG against MegaKV, KVC, KVG, and MICA by varying the read/write ratio at θ=0.5 with 8B keys and values. In addition to the competitors, KVCG latency is reported for each component (i.e., the Hot Cache and Canonical Store). When considering throughput, we include both their individual contributions along with the total.

In Figure 2.5a we note that KVC has high mean latency compared to other components. This is because the Hot Cache was designed and optimized to serve few requests. When KVC has to serve the entire data store, there is high contention which causes high latency. MegaKV has high latency because of its secondary index. In fact, with more update operations, MegaKV needs to modify the secondary index more, which causes a significant



(a) Mean Latency

(b) Throughput

Figure 2.5: KVCG and competitors varying % of read /write operations. Skew θ=0.5. 8B value sizes.

latency penalty. On a read-only workload, MegaKV has lower latency than the Canonical Store in KVCG and KVG because it has the lock-free structure of SlabHash while L-Slab requires acquiring locks.

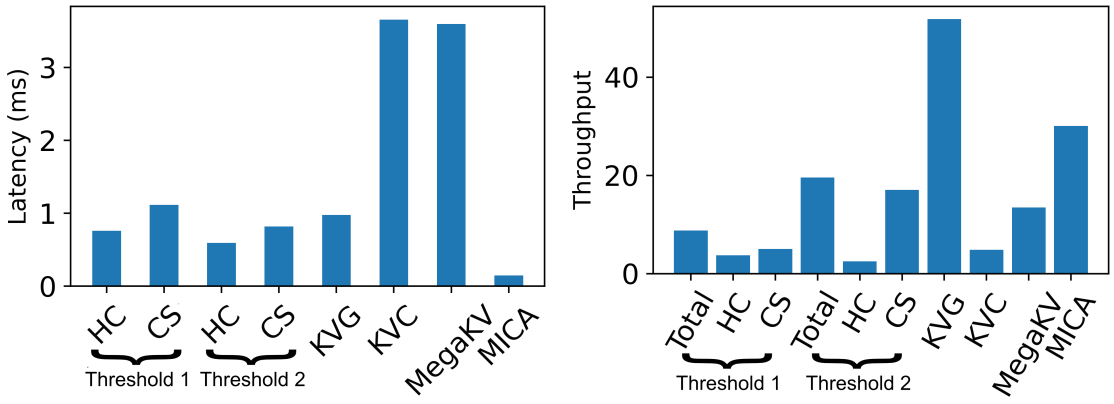
At θ=0.5, KVG and KVCG's Canonical store have similar performance. KVCG on the other hand is able to keep up with the throughput of KVG while providing low latency operations on hot keys.

KVCG's Hot Cache shows slightly lower latency than MICA (0.11ms versus 0.17ms at 80%). MICA's throughput is however lower than the GPU's. KVCG is the best choice when a throughput greater than 35.4 MOPS is needed, but an average latency below 1ms is desirable.

2.8.3 YCSB Standard Skew

In Figure 2.6 we evaluate KVCG against competitors with a traditional YCSB workload B, which consists of 95% reads and 5% updates using θ=0.99. We evaluate two thresholds: T1 is the $1.3 * 10^{-5}$ threshold we chose for theta 0.5; T2 is 0.005. Similarly to the plots in Figure 2.5, we find that KVC performs poorly. The latency is above 1ms while the throughput is low. MegaKV has similar issues to Figure 2.5, where the utilization of the

secondary index impacts the performance. MICA similarly has low throughput and low latency.



(a) Mean Latency

(b) Throughput

Figure 2.6: Performance of KVCG and competitors. Skew θ=0.99. 8B values. 95% reads/5% writes.

The most interesting change from theta 0.5 to 0.99 is that KVG performs better than KVCG at both T1 and T2. Its latency is only slightly greater than the latency of CS at T2, while the throughput is more than 2x. Although 0.99 is heavily skewed, L-Slab is able to perform well enough to achieve a large throughput when it receives enough operations to handle. At both T1 and T2, the Canonical store is unable to receive enough requests per second workload to provide significant throughput. This is due to few keys being heavily accessed at a θ=0.99. Changing from T1 to T2 yields an increase in throughput, but compared to going from a T2 (a threshold of 0.005) to KVG (a threshold of ¿1), it is unable to achieve as much of a performance gain. KVCG, however, is able to run with a lower latency on Hot Cache when compared to KVG. When using KVCG at this high a skew, a high threshold should be chosen if using a histogram model.

2.8.4 Changing the Model

One of the fundamental components of our design is the Router, which decides whether requests are to be served by the Hot Cache or the Canonical Store. In this experiment, we aim to understand the impact that our dynamic model switching protocol has on performance. Additionally, we wish to demonstrate the importance of a well-trained model.

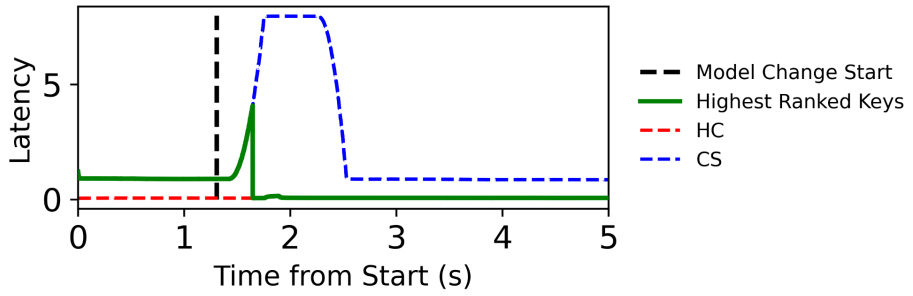


Figure 2.7: Moving average of system latency over time when changing the model.

To accomplish the above, we execute requests while training the model in the Router and then update it during execution. Initially, KVCG handles requests using a poorly trained model. At time 0s, a shadow model is trained for 5 million requests using a histogram with 10,000 bins and a threshold of $1.3 * 10^{-5}$ as previously described. Once trained, the model is installed at time 1.3s, which completes 340ms later.

Figure 2.7 shows latency as the model changes. At first, the highest ranked keys in the distribution are served in the Canonical Store because the model misclassifies requests. When the model switches, the latency of requests in the Canonical Store increases as the Hot Cache's log is replayed and compulsory-misses are enqueued for processing on the GPU. At 2.54s, these outstanding requests are completed, and the Canonical Store returns to less than 0.9ms latency. Note that once the highest-ranked keys are populated in the Hot Cache those requests are served with a latency of 0.1ms. We envision these transitions in workload and therefore the model to be infrequent similar to how Facebook's workloads exhibit high temporal locality in a short period (i.e. one hour) that decays exponentially over time [188]. We can expect the majority of the time to be spent with less than 1ms Hot Cache and Canonical Store latency.

2.9 Conclusion

We have presented KVCG, a cooperative heterogeneous key-value store designed to accelerate skewed workloads by offloading requests to less frequently accessed keys to the GPU. Our design, which includes classifying requests at runtime, effectively juggles requirements

of modern applications by providing low-latency operations on hot keys, meanwhile supporting high overall throughput by leveraging the GPU for requests on cold keys.

We have also contributed a high-performance GPU-based hashmap, L-Slab, and a memory allocation technique for Unified Memory, group allocation. Both contributions help programability and allow for the design of high performance CPU-GPU cooperative systems.

Chapter 3

Enabling High-Performance Heterogeneous Workloads Through Snapshotting and Heterogeneous Systems

3.1 Introduction

Real-time analysis of data stored in transactional data repositories is becoming increasingly important. Databases like 4Paradigm's FeDB/OpenMLDB [41, 1], PingCap's TiDB [91], Google's AlloyDB [75], Google's F1 Lightning [190], and Vegito [164] enable users to transact with databases and analyze the data concurrently. The workloads these systems process include hybrid transactional/analytical processing (HTAP) workloads and online decision augmentation (OLDA) workloads.

HTAP enables a database management system to perform both online transactional processing (OLTP) and online analytical processing (OLAP). Supporting these heterogeneous operations, which are substantially different in terms of computing demand, could lead to performance inferior to the performance of both OLTP-only or OLAP-only systems. This is mainly because accessing data while it is getting modified introduces bottlenecks within the

concurrency control due to conflicts between long-running OLAP queries and short running OLTP transactions.

To address this bottleneck, recent solutions in literature have dealt with this issue by relaxing real-time guarantees and allowing for some staleness of queries in exchange for higher performance of transactional and analytical computation [91, 164]. However, the cost of staleness in high-performance HTAP databases can be too high for some applications like credit card fraud [41] (see scenario below). Ensuring data freshness in such applications, which is informally defined as the time between the commitment of a database transaction and when its modification can be observed by other transactions, is essential to reach decisions based on up-to-date data snapshots.

The need for more stringent freshness prompted the creation of a new computing paradigm named Online Decision Augmentation (OLDA) [41]. OLDA is a specific paradigm that allows for fast feature engineering to be used in machine learning models to improve the performance of the inference task. OLDA focuses on the specific kinds of analytics related to machine learning and allows for interfacing with machine learning inference services. Although existing OLDA systems indeed achieve high performance and high data freshness (i.e., real-time order), unfortunately they relax the transactional model in order to avoid the above bottlenecks. For example, FeDB/OpenMLDB [1, 41], an OLDA database, does not allow for the use of the `BEGIN` and `COMMIT` SQL commands needed to write arbitrary transactions. As a result, performance increases, but programmability is severely affected.

The following example highlights an emerging need to maintain both freshness (through real-time order) and programmability (through general-purpose transactions that include analytics and machine learning inference computation) in modern data management systems. The example is inspired by credit card fraud detection. In a recent work [41], 4Paradigm [2], a company for banking fraud detection services, suggests that as people use credit cards, complex information stemming from the credit card transactions and their time relative to other transactions can aid in predicting fraud. In the same work, it is shown that fraud detection using machine learning inference primitives must occur within $20ms$ of the moment the transaction arrives.

Best effort approaches like those seen in recent HTAP databases [91, 164] do not guaran-

tee such freshness and low latency. On the other hand, transactional support and real-time order are necessary properties to commit a financial transaction only if the most recent snapshot of execution is found to be not fraudulent.

Programmability through expressive APIs that allow for transactional operations, atomically executed along with non-conventional computation, such as machine learning inference, is essential for engineers to work with data repositories and analyze real-time data.

Motivated by the above observations, in this chapter we introduce SnapKV, a heterogeneous data repository architecture that aims at achieving both programmability and freshness by (1) enabling transactional support along with real-time analytics, and (2) providing a general architecture to achieve high-performance transactions, analytics, and machine learning inference, all in an integrated infrastructure. The key to enabling SnapKV's heterogeneity is a new primitive called `SNAPSHOT`. This `SNAPSHOT` primitive builds off of prior work on linearizable range-queries [184, 130]. It allows programmers to perform a collection of range query operations that creates an arbitrarily large snapshot of data to be computed by OLDA and OLAP workloads. `SNAPSHOT` can be invoked within a transactional context, and SnapKV's concurrency control ensures that the snapshot remains consistent until the transaction commits.

In order to achieve all the aforementioned goals without sacrificing scalability, SnapKV follows the recent trend of integrating heterogeneous architectures, mainly graphics processing units (GPUs), in data repositories [169, 192, 53].

SnapKV's approach to heterogeneous computing is to divide transactions into smaller computational tasks and evaluate which architecture (i.e., CPU or GPU) may have more affinity to each task. We then evaluate the affinity of each task and design optimal functions/kernels for that architecture. More specifically to SnapKV, transactional primitives and data structures are more amenable to the CPU for these problems. However, when performing machine learning inference, we offload to the GPU. This type of approach to heterogeneity enables users to meet OLTP, OLAP, OLDA, and similar workload demands by utilizing high performance heterogeneous computation within their transactions.

We implemented SnapKV in C++ and tested it using enterprise CPUs and an NVIDIA Tesla V100 GPU. We compare its performance against FeDB [41, 1] for OLDA workloads,

and against Vegito [164] and TiDB [91] for HTAP workloads. The results show that SnapKV achieves up to 2.43x speedup and 45.7% lower latency compared to FeDB. In HTAP workloads, SnapKV significantly outperforms TiDB, achieving up to 604x and 4x speedup in OLTP and OLAP throughput, respectively. Compared to Vegito, SnapKV's transactional throughput is improved by 1.3x, which matches our goals of providing real-time order guarantees without sacrificing transactional performance. On the other hand, SnapKV's OLAP throughput is 16.8% of Vegito's OLAP throughput. This is not surprising due to the highly optimized column-store design Vegito adopts, which is not the focus of this chapter. Our future plans include investigating the possible extensions to include such optimizations in SnapKV.

3.2 System Overview

SnapKV is an in-memory data repository designed to concurrently handle a variety of complex workloads including OLAP, OLTP, and OLDA (machine learning). SnapKV achieves this by providing two interoperating subsystems, the *transactional* and *compute* subsystems. Our transactional subsystem implements the transactional primitives, concurrency control, and in-memory storage. Support for OLAP operations and GPUs are enabled in the compute subsystem. More in detail:

- The transactional subsystem provides transactional primitives, including `SNAPSHOT`, which are all able to support the reads and modifications needed for each workload.
- The compute subsystem is able to provide relational algebra and set operations (e.g. join or sort) along with computational kernels (e.g. general matrix-matrix multiplications needed for deep learning inference) and GPU support in order to achieve the low latency required for these workloads. The compute subsystem supports dynamic loading so that custom operations and kernels can be added to enable future workloads to achieve their performance objectives.

These two subsystems are then utilized within a transaction, allowing programmers to consistently interact with data while optimizing analysis through CPU and GPU computa-

tion. Unlike prior approaches to HTAP or OLDA, SnapKV differs in how it fuses these two subsystems into a single centralized system.

SnapKV relies on the key-value data model, which provides a useful abstraction for storing unstructured data, as well as relational data with appropriate transformations (e.g., by mapping the primary key and a representation of the rest of the record's fields to the key and the value of the key-value pair, respectively [164, 165]). Internally, key-value pairs are stored in ordered maps to support consistent range operations, as shown later.

Figure 3.1 displays how our APIs enable SnapKV to support OLAP, OTLP, and machine learning (OLDA) workloads. OLAP, OLTP, and OLDA transactions must operate atomically, consistently, and isolated. Since these transactions are heterogeneous in their computation, we identify a set of common primitive operations and augment them with the capability to run other types of computations (e.g., machine learning inference, relational algebra). Among the common primitives, we also include our `SNAPSHOT` API, which allows programmers to specify multiple keys or ranges of keys (i.e., a snapshot). Importantly, the `SNAPSHOT` API is also managed by the concurrency control to guarantee its consistency with the other transactional APIs. This snapshot can be further manipulated using other transactional APIs, but more importantly it is used by the other types of computations to atomically retrieve and analyze the target snapshot.

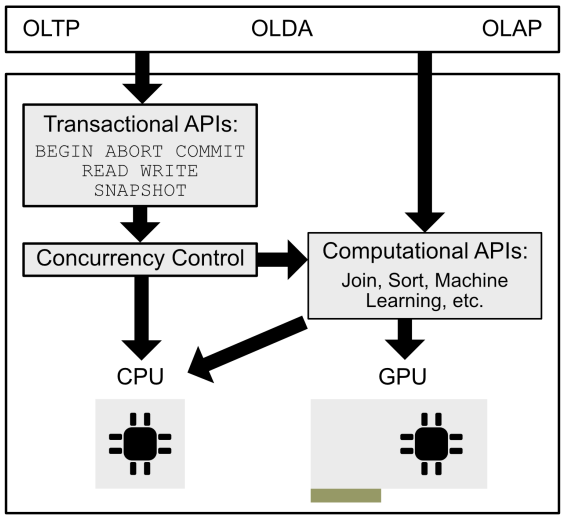


Figure 3.1: SnapKV's Heterogeneous Architecture for Heterogeneous Workloads.

At a high level, SnapKV implements a *snapshot isolation (SI)* [26] concurrency control

through multi-versioning and validation of conflicts at commit time (more details in Section 3.3). We choose to support SI because, in addition to being widely used in commercial databases [145, 172] (although often called something other than snapshot isolation [26, 78]) and widely studied [167, 38], it provides real-time order, which is necessary for our workloads. Informally, real-time order guarantees that a transaction atomically occurs between invocation and commit and that all operations that committed before in time are visible. Within snapshot isolation, a global timestamp is used as the ordering mechanism.

SnapKV operates by receiving a `BEGIN` operation, at which point the transactional manager begins a transaction. The client can issue `READ` and `WRITE` or `SNAPSHOT` operations. The APIs work as follows:

- `READ`: takes a key or a range and returns a value or list of values.
- `WRITE`: takes a key-value pair to be written or deleted.
- `SNAPSHOT`: takes a list of keys and a list of ranges and returns the corresponding list of values.
- `ABORT`: used to intentionally abort a transaction.
- `COMMIT`: used to finalize a transaction.

The `SNAPSHOT` API implements a linearizable range query over multiple ordered maps and can be invoked within a transactional context. As opposed to the multi-get API available in other data repositories (e.g., Memcached [95]), our `SNAPSHOT` performs a single optimized traversal of the data repository, which is faster and critical to support long-running code (e.g., OLAP queries or transactions that use machine learning primitives).

To support heterogeneous computation within a transaction using the GPU, we extend the transactional programming model to allow running operations to program the GPU for efficient execution (e.g., CPU/GPU memory allocation and transfer, GPU kernel scheduling). In Figure 3.1, this is denoted by the computational API. Within a transaction, the computational APIs can be invoked on data provided by a transactional API (`READ` or `SNAPSHOT`) or can be invoked on data not stored within SnapKV (e.g. arguments to the transaction). Correctness is preserved by disallowing the GPU to modify data provided by

`SNAPSHOT`, as well as synchronizing CPU and GPU execution to avoid parallelism within a transaction (more details in Section 3.4).

SnapKV does not limit its support of heterogeneous computation to what is provided within its pre-existing compute subsystem libraries; programmers can provide their own custom implementations of arbitrary operations (e.g., a linear regression task) to be run on both CPU and GPU. This heterogeneous computation can use SnapKV's primitives to retrieve data and manipulate data, consistently, with the only limitation that GPU tasks should not modify data. Customizability enables SnapKV to optimally adapt to new transactional workloads and run these operations on both the CPU and GPU.

3.2.1 Differing Approaches

The approach SnapKV takes is different from prior systems [164, 91, 41].

Figure 3.2 illustrates how existing HTAP [164, 91] and OLDA [41] systems utilize middleware or distributed protocols to couple together multiple nodes or services. This approach increases system complexity and often introduces undesirable trade-offs, such as the inability to guarantee real-time order across the system. On the other hand, SnapKV is built from the ground up and, in order to simplify system guarantees, it ensures SI across the key-value store. Furthermore, SnapKV is designed to support the computing capability of emerging hardware, such as GPUs, within transactions.

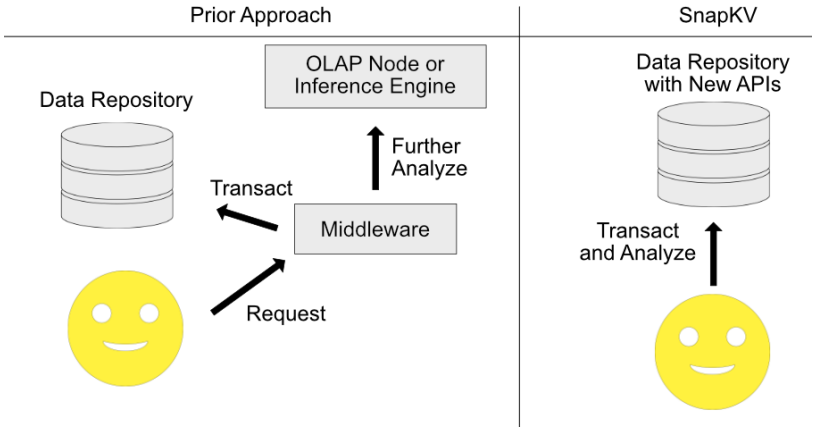


Figure 3.2: SnapKV vs Prior HTAP and OLDA Approaches.

The most basic differences in the systems can be seen when implementing a simple fraud

detection microbenchmark on FeDB [41] versus SnapKV. Within our fraud detection, we want to store information about the current credit card transaction (e.g., user, location, etc.) and retrieve/pre-process some amount of prior transactional history. At this point, we can infer whether the current transaction is fraudulent or not. A FeDB based implementation must run an insert transaction to insert a new record into a table, followed by another transaction to query the prior relevant transactions. These two operations are not atomic in FeDB. At this point, the current financial transaction and historical financial transactional information must be run through another server with an inference engine to infer.

Unlike FeDB, SnapKV does not require any networking or another node for inference. Also, this process would be representable as a single atomic operation; the transaction would begin with a `WRITE` followed by a `SNAPSHOT` of prior information. These operations would interact with the transactional subsystem and the concurrency control. Once the `SNAPSHOT` returns the relevant transactional history, the programmer will use the compute subsystem to copy the data to the GPU, run our machine learning kernels, and copy the data back. Once the compute subsystem finishes and copies the data, the programmer can handle the result of the inference and `COMMIT` through the transactional subsystem.

In Section 3.3 we detail how the transactional subsystem and the `SNAPSHOT` primitive works. In Section 3.4 we explain how the compute subsystem and heterogeneous computing is enabled within the key-value store. In Section 3.5 we detail our OLDA credit card fraud microbenchmark. Finally, in Section 3.6 we evaluate our approach in comparison to prior approaches.

3.3 Transactional Subsystem

SnapKV's transactional subsystem is designed to support the operations that are necessary to implement concurrent OLAP, OLTP, and OLDA workloads. This includes `READ`, `SNAPSHOT`, and `WRITE`. The two APIs retrieving data ensure the highest level of freshness (i.e., real-time order).

SnapKV's transactional subsystem supports SI through multi-versioning [167]. Clients are able to interact with the SnapKV transactional manager through the six operations

mentioned in Section 3.2. We determined that these operations have an affinity for the CPU instead of the GPU, and therefore we implement and optimize them for the CPU.

Algorithm 1 `COMMIT`

```
Require: Transaction tx and global timestamp gts
 1: versionLists = {}
 2: for (k, v) ∈ tx.orderedWriteSet do
 3:     l = map.getVersionList(k)
 4:     if l == null then
 5:         l = map.insertVersionList(k)
 6:     end if
 7:     lock_exclusive(l)
 8:     versionLists.append(l)
 9:     if l.latestTimestamp > tx.start then
10:         unlock(versionLists)
11:         return ABORT_AND_RETRY
12:     end if
13: end for
14: tx.commit = ++gts
15: for l ∈ versionLists do
16:     l.prepend((tx.commit, tx.orderedWriteSet(l.key)))
17:     unlock(l)
18: end for
19: return COMMITTED
```

SnapKV implements an optimistic concurrency control (pseudocode shown in Algorithm 1) that ensures the Snapshot Isolation (SI) correctness level. It works as follows. A global timestamp is read on `BEGIN`. This is stored in `tx.timeStamp`. A map associates a key with a list of versioned values. When a `WRITE` occurs, it is buffered by the transaction manager (within the `tx.orderedWriteSet` in the pseudocode). When a `READ` occurs, `tx.orderedWriteSet` is checked to see if the key has been updated; otherwise the map is read, and the most recent version with a timestamp less than or equal to the transaction timestamp is returned. Recall that, because of the SI guarantees, the return values of read operations can be avoided to be recorded for future validation. `ABORT` frees the state of the transaction, while `COMMIT` initiates the commit protocol.

The commit protocol locks in exclusive mode the keys that will be written to in increasing order to prevent deadlock. If the key does not exist in SnapKV, a new version list will be inserted and marked with timestamp 0 (i.e., no read operation can retrieve it). While

locking, the concurrency control validates that no concurrent transaction has written a newer version by comparing the transaction timestamp to the latest write of each key. If this does not hold, the transaction aborts and retries; otherwise the global timestamp is atomically incremented, updated values are written, and locks are released.

Upon commit, SnapKV must modify the in-memory storage. To implement the storage in practice, we rely on a concurrent map that implements an elementwise linearizable skiplist [93]. It should be noted that SnapKV is not limited to using this map; it is possible to use any linearizable mapping data structure that supports reads, modifications, and concurrent iteration over the data structure.

The concurrency control also supports calling GPU kernels through the compute subsystem (more details on how this interacts with the concurrency control are in Section 3.4.1).

Our main innovation to the transactional manager is the addition of the `SNAPSHOT` primitive (detailed in Section 3.3.1). This primitive is able to provide a method to perform a large series of real-time order reads, enable a programmer to quickly move the resulting data to a GPU, and help minimize the data structure's lock holding time.

3.3.1 `SNAPSHOT` Primitive

Within a data repository, there are two different types of reads that are important to support: range queries and point queries. The `SNAPSHOT` primitive implements these reads as a unified primitive.

Algorithm 2 `SNAPSHOT` primitive

Require: a set $Args$ of points and ranges
1: keys = vector({})
2: values = vector({})
3: index = vector({})
4: count = 0
5: **for** arg $\in$ $Args$ **do**
6: **if** arg is range query **then**
7: Snapshot_Range_Query(keys, values, index, count) ▷ Algorithm 3
8: **else**
9: Snapshot_Point_Query(keys, values, index, count) ▷ Algorithm 4
10: **end if**
11: **end for**
12: **return** {keys, values, index}

To call `SNAPSHOT`, a list of points and ranges is taken as an argument. The transactional subsystem will iterate through this list of arguments and perform each range query or point query. The transactional subsystem will then return the keys and values associated with each read. For example: the call `SNAPSHOT({a,range-b})` will return a list containing the value of `a` followed by the first value in `range-b`, the second value in `range-b`, and so on, along with the associated keys. There is also an index that is returned that contains the start location and end location of the ith argument in `SNAPSHOT`.

Algorithm 3 `SNAPSHOT` primitive : Range Query

```
l = map.iterator(arg.begin)                          ▷ l is a version list
for l != map.iterator(arg.end); ++l do
    lock_shared(l)
    kv = l.getKVAtTime(currentTx.start)
    unlock_shared(l)
    keys.append(kv.key)
    values.append(kv.value)
    count++
end for
iter = writeSet.iterator(arg.begin)
for iter != writeSet.iterator(arg.end); ++iter do
    if iter.key ∈ keys then
        values.update(value)
    else
        keys.append(iter.key)
        values.append(iter.value)
        count++
    end if
end for
index.append(count)
```

Algorithm 4 `SNAPSHOT` primitive : Point Query

```
kv = writeSet(arg.key)
if arg.key ∉ writeSet then
    lock_shared(l)
    kv = l.getKVAtTime(arg, currentTx.start))
    unlock_shared(l)
end if
keys.append(kv.key)
values.append(kv.value)
index.append(++count)
```

To implement this primitive in practice, SnapKV uses the multi-version concurrency

control and a concurrent map which is elementwise linearizable. Algorithm 2 details how the SNAPSHOT primitive works. The individual query is run for each argument given to SNAPSHOT. For point queries, SnapKV performs the same operations as a READ in Algorithm 4. The maps are traversed in an elementwise linearizable fashion for range queries in Algorithm 3. At each element, SnapKV acquires a shared lock on the version list and get the value associated with the timestamp of the transaction (i.e., the value with a timestamp less than or equal to the start timestamp of the transaction). When performing a range query without concurrent updates, it is guaranteed to always find the latest version consistent with the transaction's start timestamp.

If there is a COMMIT concurrent with a range query, and it increments the global timestamp before the transaction issuing the range query reads the timestamp, then all updates in the COMMIT must be observed by the range query. SnapKV guarantees this through the multi-versioning scheme along with the lock protection on the version lists. Specifically, since the COMMIT acquires all the locks before incrementing the global timestamp, the range query will be blocked if needed until the new versions are added.

In the case that the transaction issuing the range query reads the timestamp before the COMMIT increments the global timestamp, even if it will be blocked by the COMMIT it will not observe those updates because it will have a larger timestamp.

3.3.2 READ vs SNAPSHOT

SnapKV's SNAPSHOT primitive is advantageous over READ by enabling ease of copying data when doing heterogeneous programming. The structure of the vectors returned by the SNAPSHOT primitive is designed to facilitate offloading future computations to the GPU. Since it returns vectors of keys, values, and indexes, it is straightforward to move these to the GPU with three memory copies. If SNAPSHOT was not used, individual copies to the GPU must be done or the programmer must assemble the vector in their transaction and perform a similar copy. By using SNAPSHOT we are able to better support the use of heterogeneous architectures in SnapKV's compute subsystem (more details in Section 3.4.1).

SNAPSHOT can also be advantageous for performing index joins when query selectivity, i.e., the probability a key will be selected when selecting keys based on a predicate, is high.

This case occurs in `Q3` of ch-benCHmark [44], an HTAP benchmark, where there are index joins. A database must join `customer`, `order`, `neworder`, and `orderline` where the customer's state matches a predicate and then perform aggregation. Traditional query optimizers like TiDB's optimizer [91] approach this by joining `order` with `customer` and filter out customers on this join. At this point, the cardinality may be less than the cardinality of the `order` table, and the database may be able to perform the rest of the operations with fewer reads.

SnapKV can implement the above join in two ways. In both ways, it represents tables as key-value pairs with tuple `(table name, primary key)` as the key and the record as the value. When attempting to benefit from low selectivity, SnapKV would use `READ` and perform an index join, which is comprised of an $O(\log n)$ operation on the skiplist for each element in the range.

When selectivity is high, this means that the cardinality of `order` joined with `customer` after the filtering out customers is close to the cardinality of `order` joined with `customer`. This limits the benefit of approaching the query this way. Using `SNAPSHOT` instead, SnapKV will get all tables and use the compute subsystem to perform the joins. When there is minimal impact of cardinality, this means performing a $O(n)$ traversal, which is less costly than the $O(\log n)$ operations required for each element in the range when performing the index join.

In Section 3.6 we evaluate this approach to using `SNAPSHOT` and find it can lead to up to about 20% increase in throughput to approach joins this way when there is high selectivity. In order to efficiently use the primitive, the programmer must make a determination whether to perform a snapshot or individual reads. We approach the decision to use `SNAPSHOT` and `READ` based on our insights gained from testing performance versus selectivity in Section 3.6, which suggests above 80% selectivity is best for using `SNAPSHOT`. We also advise grouping any reads that follow one another as a snapshot when programming.

When implementing ch-benCHmark, we generally utilize `SNAPSHOT` on the queries and `READ` on the transactions. In the case of our credit card fraud benchmark (described in Section 3.5), we found ourselves using `SNAPSHOT` since the credit card transaction logging and detection operation has initial reads.

3.4 Compute Subsystem

The compute subsystem enables programmers to accelerate complex data analysis and operations within SnapKV by using both the CPU and GPU. In order to achieve the high performance and scalability goals of the compute subsystem, SnapKV enables GPU compute within transactions and builds upon GPU asynchronous programming APIs. Details about this are in Section 3.4.1.

Another important characteristic of SnapKV is that transactions are compiled as procedures. Within these procedures, a programmer can utilize CPU primitives (e.g., join) and tell SnapKV to dynamically load user specified kernels or functions. We detail this in Section 3.4.2.

3.4.1 GPU Compute within Transactions

Supporting GPU compute within transactions presents two problems, how the CPU and the GPU are able to share memory and how to maintain high performance when switching execution between the CPU and the GPU. Consider implementing a GPU accelerated fraud detection transactional benchmark within SnapKV. In this benchmark, historical financial transactional data are read, GPU accelerated machine learning is then used to infer whether the new transaction is fraudulent, and finalize the new financial transaction.

In order to enable the GPU to work on transactional data, the data must be accessible by the GPU and the results visible to the CPU. SnapKV relies on the standard memory copy mechanism [134] between host memory and device memory, but to retain high performance and limit overheads, it ensures minimal synchronization between the CPU and GPU. That is done by utilizing non-blocking queues of operations (called streams [82] in CUDA) to asynchronously schedule kernels for the runtime to execute.

Memory access between the CPU and GPU should be done efficiently to enable high-performance processing of transactional data. While the CPU can directly access data through the transactional manager because it is locally stored, the GPU cannot since the data structures are located in CPU memory and written by the CPU only. To overcome this limitation, the CPU should move the relevant key-value pairs to the GPU. Likewise,

any data the GPU should return to the CPU, will be written into its device memory and copied back to the CPU.

SnapKV utilizes CUDA streams, which are queues of kernel invocations and memory movement to be executed by the runtime. SnapKV's allows transactions to include CPU and GPU code blocks, which enables the invocation of CUDA stream to enqueue GPU operations. Within a CPU code block, only CPU code can be called (e.g., transactional APIs, CPU join implementations); within a GPU code block, only GPU kernels and APIs may be called within the context of the stream (e.g., deep learning kernels or memory movement). At the end of a GPU code block, the CPU is synchronized with the GPU. In our implementation, we rely on the programmer to write their transactions in this blocked manner. As an example, we implement our fraud detection benchmark as a sequence of the three code blocks. The first is a CPU block that starts a transaction and calls `SNAPSHOT` within it. This is followed by a GPU code block where the results of this `SNAPSHOT` are copied to the GPU, machine learning is run, and the results are copied back. Following the GPU code block is a CPU code block to finalize the transaction and `COMMIT`. All the code within the GPU code block occurs within the context of a single stream.

The above approach forbids parallelism between the CPU and GPU, which is important to maintain transactional semantics. In our fraud detection benchmark, for example, without synchronizing after the movement of the results from the GPU to the CPU, the CPU could miss important outcomes produced by the GPU computation. Importantly, our approach avoids synchronizing the CPU and GPU after every GPU call, which would lead to poor performance [137].

SnapKV further relies on GPU kernels to be equivalent to a single thread execution on the CPU. With this guarantee, we can reliably have the same semantics as a CPU-only data repository.

3.4.2 Compiling Transactions and Analytics

In SnapKV, transactions are written in C++, compiled as procedures that can be dynamically loaded as kernels and functions. In order to add new kernels, SnapKV compiles with the CUDA compiler `nvcc` and the C++ compiler and creates a shared object file. The

GPU kernels will be in a fat binary in the shared object file. When loading the transactions, SnapKV utilizes the operating system's existing dynamic library loading functionality. After the functions are loaded, SnapKV searches through the shared object file for the fat binary. Once SnapKV finds the fat binary the programmer wants to load, SnapKV uses the CUDA driver API to load in the kernels. The CPU primitives are added within C++ header files as inlined, and often templated functions. By inlining, SnapKV is also able to potentially help gain performance and benefit from further compiler optimization.

By compiling transactions as procedures, SnapKV enables handcrafted solutions to address specific workload requirements, as shown in our evaluation (Section 3.6).

3.5 Online Decision Augmentation Fraud Microbenchmark

Our Online Decision Augmentation (OLDA) microbenchmark is based off of the credit card fraud problem presented in [41]. Since the benchmark used in [41] is not open, we seek to create a synthetic benchmark to test OLDA, which consists of feature-engineering, machine learning inference, and an insertion into a data repository.

We design our benchmark depicting the scenario in which users perform financial transactions and a database is used to store relevant information for inferring credit card fraud. Along with each transaction, the following metadata is also provided: the latitude and longitude of the location where the transaction is performed, an identifier of the user, and an identifier of the database transaction. The microbenchmark distinguishes fraudulent and non-fradulent credit card transactions depending on their geographic location. Transactions are considered non-fraudulent if they are issued within 300km of a user's prior transaction(s).

The benchmark is parameterized by the number of users issuing financial transactions. To set up the microbenchmark, historical credit card transactions for users are generated with a uniform random distribution and initially located uniformly around four locations. When benchmarking, each credit card transaction is normally distributed around the associated location, with a standard deviation of 1 in both longitude and latitude and no co-variance between the longitude and latitude.

Each time a credit card transaction is received by a database, the following steps must occur:

- The prior credit card transaction(s) for a user is retrieved;
- The distance between the prior and current credit card transactions is calculated;
- The features: distance, latitudes, and longitudes are fed to a dense neural network trained on the data for inference;
- A new credit card transaction for a user with the associated latitude and longitude is inserted into the database.

These steps are done to test a typical OLDA workload, in which concurrent writes, reads, feature engineering, and inference are performed.

We initially train our neural network on the synthetic data set. The neural network consists of single precision floating point layers with a 5 element input, followed by a 128 neuron layer and a ReLU. This is then output into a 2 neuron layer with a sigmoid.

SnapKV performs these operations as a single transaction, with the support of GPU acceleration. Other competitors, such as FeDB [41], execute these operations as a read transaction, a separate inference process, and then an insert transaction. The absence of atomicity between these operations could impact the identification of fraudulent transactions.

3.6 Evaluation

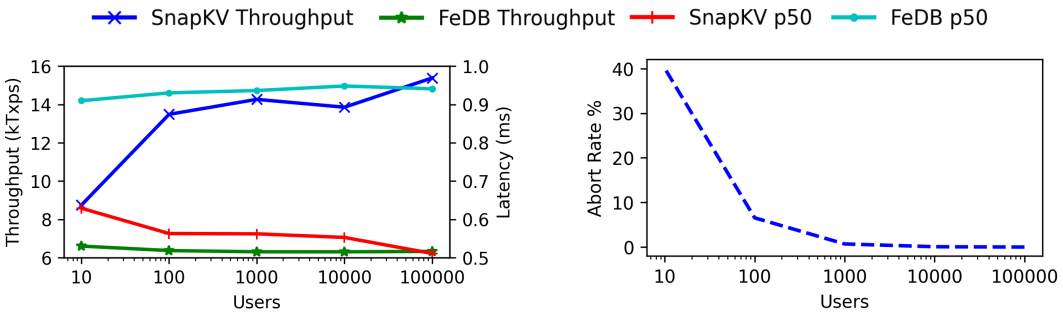
To assess SnapKV's performance under OLDA workload, we contrast SnapKV with a state-of-art OLDA database, FeDB [41, 1], using the credit card fraud detection microbenchmark described in Section 3.5. On the other hand, to evaluate SnapKV's performance under HTAP workloads, we compare it with two HTAP databases, Vegito [165, 164] and TiDB [91], using the ch-benCHmark [44] benchmark. ch-benCHmark combines the well-known TPC-C [48] benchmark with the TPC-H [49] benchmark to model an application of HTAP for a warehouse business.

SnapKV is built with GCC-10 and CUDA 11.6. We also utilize CUTLASS [100] version 2.9.0 for our deep learning kernels.

3.6.1 OLDA Workload

The test bed for our OLDA credit card fraud microbenchmark is a Cloudlab R7525 [59], which has two 32-core AMD Epyc 7542 and two NVIDIA Tesla V100. We only utilize one V100 and pin threads to one socket for the experiments. On the CPU, we fix the number of threads to 8 because in that configuration FeDB performs best in comparison to SnapKV.

When we analyze the performance of SnapKV we find that it is able to achieve 80% utilization of the GPU starting at 2 threads. Since the benchmark depends on executing on the GPU before writing back to the database, SnapKV is effectively bound by the inference computation on the GPU, and increasing CPU threads will not improve performance.



(a) Throughput and Latency when Varying Credit Card Users.

(b) SnapKV Abort Rate when Varying Credit Card Users.

Figure 3.3: Performance when Varying Credit Card Users in Credit Card Microbenchmark.

We begin evaluating SnapKV on OLDA workloads by varying the number of credit card users (Figure 3.3). In Figure 3.3a we show both the throughput and the latency. As can be seen, we consistently achieve higher throughput and lower latency than FeDB. We are able to gain increased throughput through combining both our inference and transactional systems together. FeDB utilizes HTTP requests to run queries to perform its feature engineering. This is then followed by the GPU accelerated deep neural network inference, and then a second transaction to write the credit card transaction. In contrast, we limit memory movement from system to system, and by doing so are able to achieve high performance while providing transactional semantics. We achieve between 1.32 and 2.43x speedup and

31-48% improvement in latency.

Looking more closely at the contention level in the workload, at 10 users we find that there is a significant number of conflicts, which leads to a high abort rate (40% of transactions are retried) and high system latency (0.63ms). Also, increasing the number of users decreases aborts. This trend is visible in Figure 3.3b. At 1,000 users, throughput and latency begin to stabilize as the number of users in the system increases. We find that at 10,000 users the contention level is acceptable, and so we will use this configuration in future experiments.

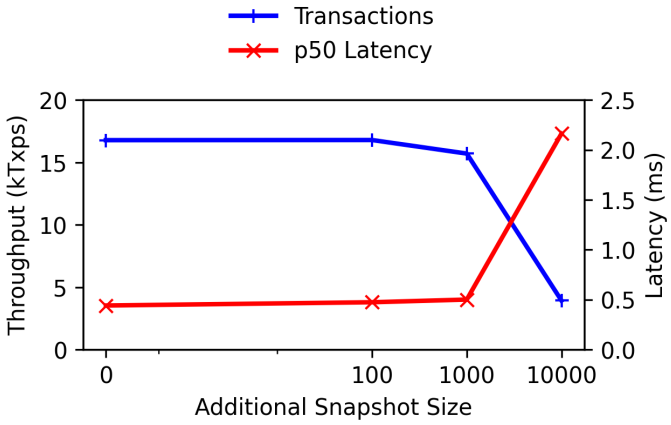


Figure 3.4: Varying Size of Additional Key-Value Pairs Snapshotted in Credit Card Microbenchmark.

SnapKV is not only able to utilize the GPU to improve performance of OLDA workloads, but it is also able to utilize the `SNAPSHOT` primitive.

We now evaluate the performance of the `SNAPSHOT` primitive. Specifically, we consider how snapshot size impacts performance, and we do this by varying the number of historical transactions retrieved. Results are shown in Figure 3.4.

When moving from no additional snapshot size to 100 extra historical transactions, we observe a 7.4% increase in p50 latency and parity with transactional throughput. This suggests that there is minimal overhead to `SNAPSHOT` in the range of 100 records. Increasing up to 1,000 reads per transaction, we find continued minimal difference in performance, maintaining 93.5% of transactional performance with a 5.6% increase in latency. It is only when we increase to 10,000 reads in a single transaction that we experience a drop in

performance.

3.6.2 OLAP and OLTP Workload

Our OLTP and OLAP competitors, TiDB and Vegito, utilize a testbed with 2 Intel Xeon Platinum 8160 CPU with hyper-threading enabled, giving us 48 cores with 96 hardware threads, and a Mellanox Connect X-5 for loopback. SnapKV utilizes the `SNAPSHOT` operation and a optimistic concurrency control to minimize the duration during which contention occurs between transactions. On the other hand, in order to achieve desirable performance for concurrent OLAP and OLTP transactions, TiDB and Vegito utilize the existing replication mechanism to execute OLAP queries in separate processes.

We test the competitors with multiple processes on a single node and rely on the loopback network interface to ensure the performance differences between all competitors are not due to the networking. We also only utilize the CPU for executing queries on SnapKV to minimize differences due to GPU vs CPU acceleration. Vegito relies on Intel TSX-NI and a Mellanox RDMA network card, so it is not possible to reproduce results on the AMD Epyc system. We reproduce TiDB's results on the AMD Epyc system and find it consistently performs better on the Intel Xeon machine, so we only compare to the Intel Xeon machine. This configuration enables Vegito to offload memory copies between processes to the RDMA network card, effectively treating the card as a hardware accelerator.

When running our experiments with ch-benCHmark, we run with 10 warehouses. The version of TiDB used is 5.4.0. TiDB is run with 3 TiKV processes, 3 placement drivers, and 2 TiDB processes. Unless otherwise mentioned, Vegito is configured to broadcast the establishment of a new epoch every 15ms, meaning that it could take longer than 15ms to establish the next interval when writes are visible on the OLAP process. We run Vegito with 1 primary process, 1 backup process, and 1 OLAP process.

To implement ch-benCHmark in SnapKV, we utilize key-value tables with primary keys mapping to rows. In this workload, we run 90% `New Order` transactions followed by 10% `Delivery` transactions. Both `New Order` and `Delivery` transaction profiles write to the `orderline` and `neworder` tables. We also concurrently run the `Q1` query profile, which reads from the `orderline` table to perform aggregation over delivered orders (orders modified by

the `Delivery` transaction). This profile creates dependencies and anti-dependencies between all transactions and queries, hence making the level of data freshness critical. We generate a uniform random warehouse and district (1 of 10 districts) for each transaction, and will read from `orderline` and `neworder` in `Q3` when a customer state matches a query. Unless otherwise stated the probability a customer is selected is 1.7%. In order to perform queries, we utilize single threaded transactions. Instead, TiDB and Vegito parallelize parts of their queries. Unlike our competitors, we are able to run transactions and analytics with real-time order guarantees using ch-benCHmark.

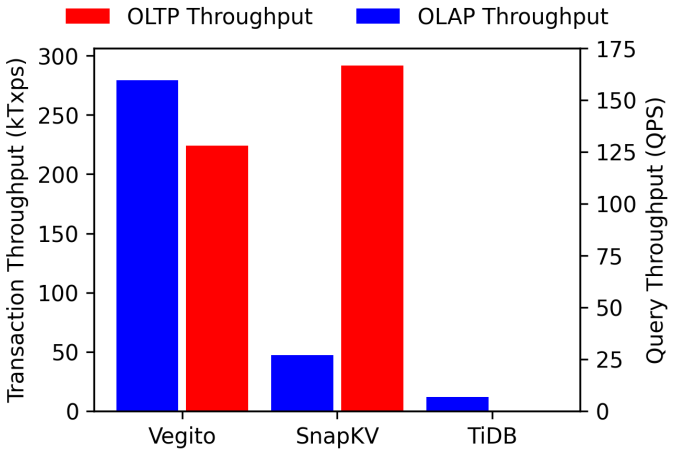


Figure 3.5: Throughput of different HTAP Solutions.

In Figure 3.5, we run our custom `New Order`, `Delivery`, and `Q1` workload. As also confirmed in [164], TiDB does not perform as well as in-memory systems due to the disk access costs (to minimize that, our testbed is equipped with SSD drive). SnapKV is able to achieve a significant improvement in transactional performance (up to 600x) and a 4x improvement in query performance.

Compared to Vegito, SnapKV obtains 1.3x the transactional throughput, but it has a lower query throughput, up to 16.8% of Vegito's. The improvement in transactional throughput comes from the minimized concurrency control overhead: SnapKV does not need to log to a backup node upon commit, while Vegito does. On the other hand, Vegito is able to perform better in terms of query throughput because of two reasons. First, it works on stale data, therefore no synchronization with the ongoing workload is required;

second, many of its components are optimized for a column store relational database. As opposed to Vegito, SnapKV guarantees real-time order and does not focus on implementing a column store. Future work could be spent considering this optimization.

In our next experiment (results in Figure 3.6) we dig deeper in the performance impact of allowing stale accesses. We exclude TiDB because of its comparatively poor performance.

Vegito maintains an oracle that periodically broadcasts the establishment of a new epoch in which the analytical database apply batch updates. In this experiment, we vary the epoch time. We calculate freshness as the number of epochs difference between the OLAP node and the OLTP node at any given time. To achieve real-time order, the epochs would need to be consistently equal.

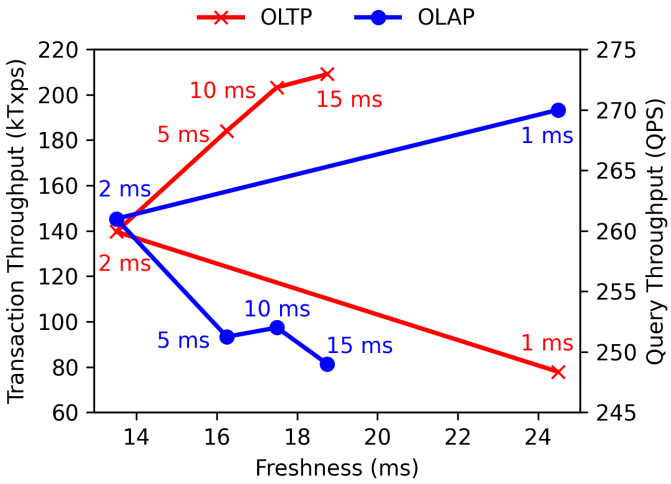


Figure 3.6: Throughput versus Freshness of Vegito While Varying Epoch Timing (Epoch Times Written on Plot). SnapKV has 0ms freshness, 292 kTxps, and 26.9 QPS.

By varying the epoch time, we find Vegito is unable to reach real time order (0ms). Vegito is only able to achieve a minimum freshness of 13.5 ms when the epoch time is set to 2ms. When running at its most fresh, Vegito is only able to perform 66.8% of the maximum transactional throughput and 96.7% of the maximum query throughput it can achieve.

Moving from a 15ms epoch to a 2ms epoch, Vegito is able to improve freshness, however, query throughput increases and transactional throughput decreases. That is because, decreasing the delays between epochs entails the transactional subsystem needs to push all logs to the backups at higher rate. As a result, more synchronization is needed to log

updates and transactional performance suffers. Interestingly, in this case the OLAP process is able to serve more requests while waiting for updated logs to be pushed. This increases the query throughput and keeps it stable; query throughput only varies between 249 and 270 qps while transactional throughput varies between 78 ktxps and 209 ktxps.

Vegito is unable to maintain its freshness when moving from a 2ms epoch to 1ms epoch. At this point, logging to the OLAP process is a significant overhead for the OLTP process. This overhead leads to a significant skew between the epoch that the OLAP process has stored and the epoch that the OLTP process has stored, which makes the OLAP transactions more stale. Unlike Vegito, SnapKV is able to maintain consistency for both the transactions and queries without significant impact on their performance.

Next, we evaluate the performance of the `SNAPSHOT` primitives versus the `READ` primitive when changing the selectivity of queries (i.e., the probability a key is selected). When using `READ`, we perform index joins and utilize the selectivity of the query to minimize the number of reads that must occur of the following tables. When using `SNAPSHOT`, we snapshot all the tables, rebuild indexes, and perform the joins. In Figure 3.7, we evaluate changing the selectivity of ch-benCHmark's `Q3` query by changing the proportion of customers we filter out by selecting on the customer state when implementing with `READ` and when implementing with `SNAPSHOT`.

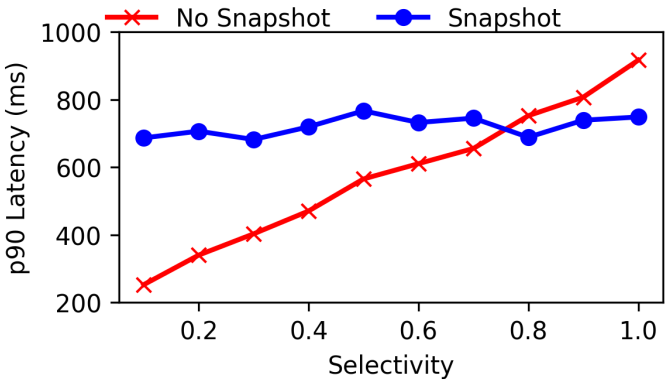


Figure 3.7: Read Versus Snapshot (p90 Latency) While Varying Selectivity of the `Q3` Query.

At a selectivity of 0.1, the query throughput when using `SNAPSHOT` is 34% of using `READ`s and the p90 latency is 2.7x that of using just `READ`s. At 0.8 selectivity and above, we find that `SNAPSHOT` becomes a better option to achieve high performance transactions. It is able

to achieve 6% greater throughput and 8.5% of the p90 latency. This is because we are able to perform all reads in a sequence, which limits time for contention. We also are able to benefit from having a copy of data, which we can reuse rather than needing to perform reads on the same key multiple times. In the best case, using `SNAPSHOT` can result in 20.4% greater throughput and a 18.3% decrease in the p90 latency.

3.7 Related Work

Both academia and industry have been interested in the prospect of combining various analytical workloads and transactions in a single data repository [107, 168, 119, 91, 164, 41, 14, 132, 3, 101]. MemSQL [168], SQLServer [119], TiDB [91], Vegito [164], and FeDB [41] support these kinds of workloads. These analytics and transactions are not limited to relational or key-value models, and include graph processing [160, 176] and in-database machine learning [41, 1].

Recent innovations in HTAP databases have focused on using replicas to serve queries over fresh data [91, 164, 3]. TiDB and Vegito are recent examples of utilizing the availability mechanism to replicate state onto a query processing node. Unlike these solutions, we only rely on a single node/process to provide support for queries and transactions.

Traditional solutions for HTAP focus on multiversion concurrency control. Diva [101] focuses on supporting HTAP in disk based MVCC databases. HyPer [132], RateupDB [107], Vegito [164] and others utilize aspects multi-versioning as well. We similarly focus on utilizing multiversioning, which enables the optimistic approach to concurrency control that we rely on in SnapKV.

Other HTAP database solutions, such as RateupDB [107] have focused on using an alpha and delta store model, where transactions modify the delta storage and are merged over time into the alpha store. RateupDB also focuses on the prospect of using the GPU for analytical processing similar to SnapKV. Unlike SnapKV, RateupDB is specifically designed for the relational model, relational algebra, and column storage. SnapKV takes a more general approach, allowing for any computation on the GPU within transactions.

Key-value stores enable a simpler data model and have been greatly explored. Re-

cent publications have focused on enabling key-value stores to benefit from new memory architectures, such as how ChameleonDB [195] is capable of utilizing persistent memory. EvenDB [72] focuses on exploiting spatial locality within key-value workloads to enable high-performance. SnapKV neither benefits from new memory architectures nor exploits spatial locality, but these techniques are complementary to SnapKV.

Even more related papers on key-value stores focused on the prospect of GPU computing. KVCG [53] provides a method to serve skewed key-value store workloads by cooperatively utilizing the CPU and GPU. MegaKV [192] provides another method for serving key-value store workloads using the GPU as an accelerator. Unlike KVCG and MegaKV, we focus on transactional workloads and must ensure atomicity, isolation, and consistency across more than one operation. We also do not utilize the GPU to do storage, only computation.

Related key-value store works enable multi-get operations, where a single request can read multiple keys in an elementwise-linearizable fashion. Key value stores such as Memcached [95] and KVCG [53] enable multi-get operations. `SNAPSHOT` is similar to multi-get in that it is able to get multiple keys by providing a set of keys or range. Unlike multi-get, `SNAPSHOT` operations are used within transactions and guarentee consistency between all of the reads in the snapshot. `SNAPSHOT` is designed as an API for optimizing read-only and read-heavy transactional workloads.

For the programability of our system, we rely on existing libraries to implement queries. We utilize C++ and the CUDA programming language [134] and provide support for libraries like CUTLASS [100] to design of deep learning solutions. We demonstrate the ability to run deep learning inference in our key-value store. We envision other CUDA libraries may be of use in our system, and could be easily used within SnapKV. Related to the deep-learning aspects of our work, Han et al. [80] present a GPU preemption methodology to improve throughput for inference workloads with real-time deadlines. This work is complementary to our work. Our methodology does not focus on preemption and instead focuses on evaluating the feasibility and performance of using the current CUDA runtime to compute within a key-value store.

Our closest related works support both transactions and analytics. TiDB [91] and Vegito [164] use the availability mechanism to run analytics on a separate node. TiDB performs

best on fast storage like NVMe. Vegito, however, is in memory and utilizes DrTM+H [182] to support transactional workloads with remote direct memory access (RDMA). The backup analytical node is based on MonetDB [92]. Both TiDB and Vegito have a staleness issue (order of tens of *ms* in Vegito), which makes it more difficult to reason about the system, and prevents these systems from being broadly applied.

FeDB [41] is a closely related custom solution for feature engineering and machine learning inference. It is designed in contrast to HTAP solutions like TiDB and Vegito in how it needs even fresher data. Similar to FeDB we support machine learning and provide the programmer with the freshest possible state. Unlike FeDB, SnapKV is designed as a general solution, not just a solution for online decision augmentation workloads.

3.8 Summary

We introduced SnapKV, a transactional data repository architecture for heterogeneous workloads (OLTP, OLAP, and OLDA) that guarantees real-time order without sacrificing performance. This is achieved through the use of the new `SNAPSHOT` API and by supporting heterogeneous devices. Our evaluation confirms that SnapKV is able to outperform state of the art competitors in both OLDA and HTAP workloads.

Chapter 4

Semantic Transactional Processing

4.1 Introduction

Blockchains are becoming an increasingly prevalent secure distributed transactional processing scheme for large-scale systems. Unquestionably, its popularity comes from its ability to implement payment systems [185, 127, 115] and possibly enable the establishment of supports for central bank digital currency [178, 104]. In order to execute distributed transactions in a typical blockchain, *1)* a proposer pulls transactions from the mempool (i.e., a transaction repository) in some order to form and validate a block before proposing it; *2)* a consensus algorithm is then responsible for agreeing on that block; *3)* and validators receive that block and execute it deterministically to validate and replicate the blockchain state, ensuring immutability and irrefutability. As will be clear later, in this chapter, we provide a transaction processing scheme that proposers can use before submitting the block to consensus and validators during consensus to maximize the number of transactions committed in a block.

We want to focus on two important characteristics of the above process. First, the outcome of processing transactions in a block must be deterministic so each blockchain node can reproduce the shared state after executing a sequence of blocks. Second, many blockchain implementations, especially those permissionless, focus on financial transactions (i.e., moving money between accounts) [167, 66, 170, 10] and have a rigid programming

model[1].

The high performance of the concurrency control that ensures the deterministic transaction processing schema is essential for implementing competitive blockchains. Prior work [71, 123, 149, 153] has considered various ways to parallelize this execution. These techniques must detect and resolve conflicts (e.g., anti-dependency [5] or blind write [102]) to process transactions in a way that is equivalent to running them sequentially in the pre-defined order. The overhead entailed by these advanced synchronization schemes often nullifies the benefits of the parallel computation [158, 175, 152]. To address this problem, we *a)* leverage the strict programming model of blockchain implementations and the absence of interactive transactions, and *b)* effectively exploit the commutativity property of transactional operations to improve performance.

Because of *a)*, it is often possible to define compensating transactions [103, 116] that annul the effect of already committed transactions. This approach would allow us to avoid instrumentation, which is otherwise needed to keep track of transaction dependencies. Regarding *b)*, concurrency control designs integrating commutativity for performance's sake have shown very limited practicality when dealing with general purpose transactions [70, 105]. However, in the context of blockchain, semantics can play a critical role in achieving high performance while processing transactions deterministically. In fact, commutative transactions are equivalent if serialized in any order and enable us to process transactions in parallel without worrying about conflicts. On the other hand, the identification of commutativity is simply done by verifying the accesses to the involved accounts in the transactions, which are already available before execution.

In a nutshell, our novel concurrency control, *OranguTAN*, works as follows. It decomposes each transaction into a sequence of atomic operations and executes multiple transactions as if they commute (fast path). In the case a violation in the expected semantics is detected, we compensate the misordered transactions and fall back to a directed acyclic graph (DAG)-based approach that deterministically defines an order among all transactions that still need to be finalized. Building a graph with transactions' dependencies and analyz-

[1]We do not consider more complex programming models, such as smart contracts [9, 171, 33, 123] because they often allow for general-purpose transactions that hamper the applicability of our semantic-based optimizations [159].

ing it introduces overheads often unsustainable for online transaction processing [167, 151]. We circumvent this issue by designing an efficient algorithm that takes advantage of the massive parallelism of Graphics Processing Units (GPUs).

GPUs have been used within blockchains to accelerate computation, including proof of work [64] and zero-knowledge proofs [114, 55, 54, 179]. GPUs, in comparison to CPUs, have significantly more cores with a smaller set of instructions, which can perform tasks in a single instruction multiple thread (SIMT) paradigm versus a multiple instruction multiple data (MIMD) paradigm of CPUs. We use these architectural characteristics to design an algorithm that efficiently builds the DAG on the GPU and passes it to the CPU for processing.

This chapter focuses on OranguTAN as a standalone component for transactional processing. However, when put in the context of a blockchain deployment, OranguTAN's protocol can be used by both blockchain proposers, as they form and validate the block before submitting it to consensus, and by blockchain validators, as they validate and process it during consensus. Effectively, in this case, OranguTAN leverages the simple semantics of blockchain transactions that transfer money between accounts to offer a deterministic *reinterpretation* of the sequential order of the block.

We implement OranguTAN in C++/CUDA and compare against a concurrency control that is designed for commutative transactions, SPEEDEX [153], and two concurrency controls that do not make use of commutativity, Block-STM [71] and Strife [149]. We utilize two blockchain-inspired microbenchmarks, SmallCoin and SmallCoin+. We find OranguTAN's approach leads to a greater than 17x improvement over Block-STM and a greater than improvement over 26.5x over Strife due to its use of commutativity and an efficient resolution of conflicts in the fallback path. Compared to SPEEDEX's enforced commutativity approach, OranguTAN achieves a 2x performance improvement due to its better use of atomic operations. We further find that utilizing the GPU enables OranguTAN to achieve 5.8x the transactional throughput in the fallback path.

The rest of the chapter is structured as follows. Section 4.2 describes the blockchain model we assume. In Section 4.3, we overview the OranguTAN concurrency control and programming model. In Section 4.4 we first characterize blockchain transactions and demon-

strate why we can optimize for their commutativity. In Section 4.5, we discuss the algorithmic approach to solving this problem through our OranguTAN algorithm and the DAG-based fallback. In Section 4.6, we discuss how we are able to exploit the heterogeneous nature of the blockchain when utilizing this algorithm. Finally, in Section 4.7, we evaluate the performance of our algorithmic approach. In Section 4.8, we finally discuss related work.

4.2 Reference Blockchain Model

We consider a blockchain model where transactions operate on accounts, and the blockchain maintains account balances rather than tracking unspent transaction output, as done by Bitcoin [127]. Our assumed model follows that of Ethereum [185] and other chains [67, 170], including blockchain-based decentralized applications such as Uniswap [4].

The set of transactions utilized in the blockchains adopting our model comes from a service called a mempool, which is created through a gossip protocol. The mempool contains transactions that clients to the blockchain would like to be executed. Proposers in the blockchain may pick any set of transactions and propose them through a consensus protocol. The choice of consensus protocol is orthogonal to OranguTAN, and other competing transaction processing approaches.

Unlike more traditional concurrency control protocols (e.g., 2PL [63]), we require determinism to replicate the shared state across the blockchain nodes correctly. Deterministic transactional processing guarantees that all nodes executing the same block of transactions will have an identical final state.

4.3 Overview of OranguTAN

Our concurrency control, Optimistic semantic-aware TrANsaction processing, OranguTAN, is designed for blockchain systems. Some of these systems have a rigid programming model (described in detail in Section 4.4) where money is exchanged between accounts through simple transactions that involve increasing and decreasing values (e.g., balance and/or some metadata such as a sequence number [66]) associated with various accounts. For simplicity,

we name this model *send-receive-money* (or SRM in short). Because of the nature of SRM, it is possible to define compensating actions that revert the effect of an unwanted commit. Some other blockchains, such as [170, 10, 66], offer a Turing-complete smart-contract feature; however, they still have use cases in which a substantial fraction of transactions fit the SRM model. It may be possible to extend our model for use cases with a mixture of SRM and non-SRM transactions, especially if SRM transactions dominate the workload; we leave this as future work.

We utilize the semantics of SRM to break down each transaction into a sequence of atomic operations and run the atomic operations of different transactions in parallel. When we recognize a transaction's execution is no longer equivalent to its original semantics (e.g., funds were not available to perform a transfer in the first place), we undo it by executing compensating actions and re-execute it by relying on a graph-based concurrency control, optimized for GPU. As it will be clear later (Section 4.5), OranguTAN's technique to execute transactions, paired with the fact that transactions can be arbitrarily selected from the mempool, results in the generation of a schedule that:

(i) can be leveraged to define the order of transactions in a block before its proposal to the consensus layer and

(ii) can be deterministically and efficiently (using OranguTAN) reproduced by replica nodes after consensus is established.

In the following subsections, we divide the explanation of our concurrency control into two paths: the fast path and the fallback path.

4.3.1 Fast Path

In the SRM programming model, transactions follow the profile in Algorithm 5, where the money is sent only if the sender has a great enough balance. This model allows for many transactions to commute, as explained below. Two transactions commute when the state of the data repository after running transactions T_1 and then T_2 is equivalent to running T_2 and then T_1. Trivially, two non-conflicting transactions commute, but other transactions in this model can also commute. Consider three users: Alice, Bob, and Cindy. If Alice

has a great enough balance, she can run two transactions within a block to send money to Bob and Cindy in either order; the final state of Alice's, Bob's, and Cindy's accounts is equivalent.

Suppose we allow these conflicting yet commutative transactions to interleave. In that case, it is enough to execute each withdrawal atomically and in parallel and then execute all deposits atomically and in parallel to produce a schedule where no account ends with insufficient funds to perform all the transfers in the block. When Alice's transactions are run, the blockchain can deduct the money to be sent to both Bob and Cindy first and then increment the balance of Bob and Cindy's accounts. The outcome is equivalent to performing the transaction with Bob first and then with Cindy.

Even in a blockchain, transactions do not always commute. In our example, Alice might not have enough funds to send to Bob and Cindy but enough availability to complete one of these transfers. In this case, the order of transactions will determine which transaction succeeds and which aborts. It is important to note that unavailability of funds is the *only* reason for a transaction to abort in OranguTAN. Because of this, our concurrency control in the fast path does not need to keep track of accesses performed by transactions. When an abort occurs in the fast path, OranguTAN must ensure that its modification to the sender's account is undone before re-execution. That is because transactions are broken down into multiple atomic operations, and therefore, their effects are immediately applied to the shared state.

Recall that many client transactions are batched into a block in the blockchain, no transactions are interactively submitted, and no output is externalized to clients unless the entire block is committed. That means, even if a transaction applies its modifications to the shared state and then aborts, no other transaction but those in the block can access the change. As a result, by tracking transactions' dependencies, undoing the effect of a transaction that aborts due to the unavailability of funds using cascading rollback prevents final inconsistent states. The fallback path ensures this.

4.3.2 Fallback Path

The fallback path is designed for handling non-commutative transactions. We consider the block's ordering as a reference order for conflicting transactions and build a directed acyclic graph (DAG) to order them accordingly. Recall that there is a conflict between two transactions if both access the same data and at least one writes it concurrently. Because of the simplicity of the SRM model, account IDs are provided by the programmer and are not defined at runtime. With this assumption, we can identify transactions' access sets (i.e., their read-set and write-set) by parsing the transaction APIs for reading and writing accounts.

We move from the fast path to the fallback path if there are aborts. The set of aborted transactions is given to the fallback path. Compensating transactions enable us to undo the modifications performed by these aborted transactions efficiently. One transaction, T_2, compensates another transaction, T_1, if running the transaction T_1 and then T_2 results in the same state before executing T_1. In the running example, if Alice sends 10 credits[2] to Bob, it can be compensated for by having Bob send Alice 10 credits. Note that compensating transactions are guaranteed to be committed. In fact, compensating for an aborted transaction means depositing money, which is an operation that always succeeds.

Not only are the transactions aborted in the fast path undone (through compensation), but also all their dependent transactions, as identified in the DAG. Compensated transactions are then processed in the fallback path, following the order imposed by the DAG. Non-conflicting sets of transactions in the DAG are not connected, and each subgraph that is not connected can be executed in parallel. Transactions in a subgraph are executed sequentially to avoid conflict resolution, but since they can still abort due to the unavailability of funds, their writes are buffered until the transaction's completion.

Although the DAG-based schema is well known in literature [167, 151], the novelty of our approach lies in using it as a fallback for non-commutative transactions and redesigning it to take advantage of the GPU parallelism. OranguTAN can utilize the GPU to build an upper triangular matrix that represents this DAG efficiently. This process is accomplished

[2]We abstract the unit of exchangeable money with credits.

in parallel while the CPU executes the fast path. OranguTAN deploys GPU-specific optimizations for this algorithm, as detailed in Section 4.6.

4.3.3 Example Flow

We now show the execution flow of OranguTAN through a simple example. Let us consider the block of transactions in Table 4.1. The block contains subsets of transactions that are non-conflicting and subsets that are conflicting.

Table 4.1: Example Block.

Transaction	Read Set	Write Set
T_1: A sends 5 credits to B	$\{A, B\}$	$\{A, B\}$
T_2: X sends 2 credits to Y	$\{X, Y\}$	$\{X, Y\}$
T_3: A sends 5 credits to B	$\{A, B\}$	$\{A, B\}$
T_4: C sends 5 credits to D	$\{C, D\}$	$\{C, D\}$
T_5: D sends 5 credits to C	$\{C, D\}$	$\{C, D\}$

Consider the following initial state for the blockchain: A, D, and X start with a balance of 10 credits, and B, C, and Y start with a balance of 0 credits.

When we execute these transactions, the fast path will enable operations within the transactions to interleave. For example, the following ordering of operations may occur:

1. T_2 : A decrements 5 credits from its account.
2. T_1 : A decrements 5 credits from its account.
3. T_3 : X decrements 2 credits from its account.
4. T_4 : C decrements 5 credits from its account.
5. T_4 aborts because the balance of C is now negative.
6. T_5 : D decrements 5 credits from its account.
7. T_3 : Y increments 2 credits from its account.
8. T_3 finishes.
9. T_5 : C increments 5 credits to its account.
10. T_5 finishes.
11. T_1 : B increments 5 credits to its account.
12. T_2 : B increments 5 credits to its account.
13. T_1 finishes.

14. T_2 finishes.

While executing these transactions, the fast path keeps track of which transactions abort for use in the fallback path. If no transaction aborts, the originating block can be considered successfully committed without processing in the fallback path.

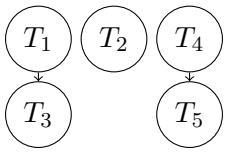


Figure 4.1: DAG of Transactions.

While the fast path executes, the GPU creates the DAG of those transactions. The DAG will look like the one in Figure 4.1. The DAG captures the information that T_3 is ordered after T_1 within the block, and the two have transactional dependencies. It also captures similar dependencies for T_4 and T_5. The graph further demonstrates that the subgraphs containing T_1, T_4, and T_2 do not contain transactional dependencies between them.

When in the fallback path, OranguTAN knows that T_4 was aborted in the fast path. This means that T_5 may have operated on an incorrect state and must also be rolled back. We utilize compensating transactions to undo the actions of T_5 and T_4. Now we execute the transactions in order, where T_4 will abort and T_5 will finish. The final state is equivalent to the serial schedule described in Section 4.5.2.

Note that, in the example above, all decrement operations precede all increments. This is not coincidental. As will be clear in the next sections, transactions in the fast path implement a logical barrier between all increment and decrement operations performed by the transactions in the block. This is done to avoid the final transaction execution order not being equivalent to the sequential one.

4.4 Reinterpreting Blockchain Transactions

The programming model of popular blockchains (e.g., Ethereum [185], Stellar [115], Aptos [66]) fits our send-receive-money (SRM) programming model, where money is sent and received between accounts. As discussed earlier, this account-based model differs from Bitcoin's unspent transaction output model [127], where transactions can only spend unspent

currency from prior transactions.

In addition to that, another important operation that practical blockchain implementations account for is the withdrawal of a monetary amount, often called *gas fee*, charged from the account performing the transaction as a fee for utilizing computing resources on the blockchain. Blockchains can process a transaction that will cause an overdraft by committing the gas fee if there is enough money to cover it [66]. For example, if Alice tries to send Bob 10 credits, the fee is 1 credit, but Alice only has 1 credit, the blockchain will take the 1 credit from Alice for the gas fee but not send the money to Bob. This structure means that sometimes only a portion of the transaction (i.e., the gas fee) is committed, and other times, the entire transaction is committed.

It is important to consider that we do not model other operations, such as balance checking, that do not occur on-chain (i.e., within a blockchain transaction). These operations can be performed locally on any node replicating the blockchain state.

We first characterize a simple example that we call *SmallCoin* after its similarity to the well-known SmallBank benchmark [11]. Next, we consider the more complex SRM programming model that includes the abstraction of gas fees (generalized from the programming model of the Aptos blockchain [67]). We name this *SmallCoin+*. We take both transactional profiles and reinterpret them into two parts consisting of atomic operations as described in Section 4.3. First, we perform all decrements of currency (Part I), followed by all other operations (Part II). In OranguTAN, transaction's Part I returns `true` on success and `false` on abort. Then, Part II is processed, knowing whether Part I succeeded or aborted.

Algorithm 5 SmallCoin.

```
1: if sender.bal ≥ amt then                ▷ amt is transaction amount
2:     sender.bal -= amt                   ▷ bal is balance
3:     receiver.bal += amt
4: else
5:     abort()
6: end if
```

Algorithm 6 SmallCoin Part I.

```
1: return sender.bal.atomic_sub(amt) ≥ 0
```

Algorithm 7 SmallCoin OranguTAN Part II.

```
1: if not aborted then
2:     receiver.bal.atomic_add(amt)
3: end if
```

4.4.1 SmallCoin without gas fee

SmallCoin consists of a transaction profile, see Algorithm 5, that sends money from one account to another. A SmallCoin transaction has two final states:

- Commit: Funds are sufficient, and money was sent.
- Abort: Funds are insufficient, and money cannot be sent.

If all transactions commute (i.e., all sender accounts have a great enough balance), they can be trivially rewritten to use atomic subtraction/addition operations, as in Algorithms 6, 7, 9, and 10. Relaxing transactions' isolation by using atomic operations allows us to design a fast path where all transactional operations safely complete, skipping any instrumentation and its associated overhead.

4.4.2 SmallCoin+: SmallCoin with gas fee

Algorithm 8 SmallCoin+.

```
1: gas_amt = sender.gas_cost
2: if sender.bal ≥ gas_amt then
3:     sender.bal -= gas_amt
4:     sender.seq_num += 1                          ▷ seq_num is the sequence number
5:     if sender.bal ≥ tx_amt then
6:         sender.bal -= tx_amt
7:         sender.sent_events += 1                  ▷ sent_events tracks sends for an account
8:         receiver.bal += tx_amt
9:         receiver.rec_events += 1                 ▷ rec_events tracks receives for an account
10:    end if
11: else
12:    abort()
13: end if
```

SmallCoin only has two potential final states: committed or aborted. However, production blockchains can have multiple final states when considering additional transactional

Algorithm 9 SmallCoin+ Part I.

```
1: gas_amt = sender.gas_cost
2: tot = gas_amt + tx_amt
3: sender.sent_events.atomic_add(1)
4: return sender.bal.atomic_sub(tot) ≥ 0
```

Algorithm 10 SmallCoin+ Part II.

```
1: if not aborted then
2:     sender.seq_num.atomic_add(1)
3:     receiver.bal.atomic_add(tx_amt)
4:     receiver.rec_events.atomic_add(1)
5: end if
```

operations, such as the well-known gas fee (Algorithm 8). These transactions can end in three states:

- Abort: Funds are insufficient.
- Committed-Only-Gas-Cost: Funds are only sufficient to deduct the gas fee.
- Committed-Send: Funds are sufficient to send money and deduct the gas fee.

In order to translate this into atomic operations and compensating transactions, the transaction becomes Algorithm 9 and 10 with the compensating transaction, Algorithm 11.

In our concurrency control, some modification of the transaction is needed to handle transactions with more than two final states. In the case of SmallCoin+, we combine the subtraction of the gas amount and transaction amount into a single atomic subtraction. Thanks to the atomicity of this subtraction, we can handle cases where two transactions with the same sender occur, and one ends in the Committed-Only-Gas-Cost state (i.e., could not commit in the fast path) while the other ends in the Committed-Send state (i.e., finished in the fast path). The validity of these transactions' outcomes is then verified in the fallback path, matching their order with the order defined by the DAG.

4.5 Concurrency Control

OranguTANs read and write shared data into an in-memory key-value map. As mentioned in Section 4.4, in many cases, blockchain transactions commute. However, when blockchain

Algorithm 11 SmallCoin+ compensating transaction.

```
1: gas_amt = sender.gas_cost
2: tot = gas_amt + tx_amt
3: sender.bal.atomic_add(tot)
4: sender.sent_events.atomic_sub(1)
5: if not aborted then
6:     sender.seq_num.atomic_sub(1)
7:     receiver.bal.atomic_sub(tx_amt)
8:     receiver.rec_events.atomic_sub(1)
9: end if
```

transactions do not commute, they will self-abort (i.e., a sender with an insufficient balance will cause an abort). Because of the simple nature of SRM, this abort can only occur when funds are withdrawn, which is the first part of an OranguTAN transaction. Since transactions' writes are not buffered but applied directly in the fast path, these writes must be undone. We log these transactions to later undo them through compensating actions that roll back their changes (i.e., the funding withdrawn).

Our approach, OranguTAN, runs transactions in the fast path by converting transactional operations into two parts consisting of atomic operations. This transformation inherently handles race conditions on concurrent writes to the same accounts. When a transaction self aborts, which is the only possibility in OranguTAN since conflicts are not tracked and therefore the concurrency control engine cannot forcefully abort an incorrect execution, it is possible that the transaction cannot commute with other conflicting transactions. To handle this case, the concurrency control uses a directed acyclic graph (DAG) to construct a partial order where non-conflicting transactions are executed in parallel while conflicting transactions execute sequentially (the fallback path).

Algorithm 12 Fast Path.

```
1: aborted_set = {}
2: parallel for tx in block do
3:     aborted = execute_transaction_pt_i(tx)
4:     if aborted then
5:         aborted_set.insert(tx)
6:     end if
7: end parallel for
8: parallel for tx in block do
9:     execute_transaction_pt_ii(tx, tx ∈ aborted_set)
10: end parallel for
```

Algorithm 12 details the fast path. We parallelize each part of the execution of transactions in a block. We execute the first part of the transactions with atomic operations (e.g., fetch and subtract to send or receive, as in Algorithm 6 and Algorithm 9). We record any conflicting and non-commutable transactions that abort. Then, we execute the second part of the transactions (e.g., Algorithm 7 and Algorithm 10). We pass the set of aborted transactions to the fallback path.

4.5.1 DAG-based Fallback Path

We use the traditional conflict graph representation for transactions [27], meaning a DAG where transactions are vertices and edges represent read-write, write-read, and write-write conflicts. In order to deterministically execute transactions in parallel, we order those with conflicts by following their sequence order in the originating block from the blockchain.

As an example, let us consider two transactions T_1 and T_2 where T_1 is ordered first in the block and:

- T_1: A sends money to B (read/write A, read/write B).
- T_2: C sends money to A (read/write C, read/write A).

T_1 and T_2 have potential write-write, read-write, and write-read conflicts on A's balance. This would lead to a DAG where the T_1 node has an edge to the T_2 node.

If we add a set of transactions, S, that does not conflict with T_1 or T_2 (e.g., transactions operating on accounts X and Y), we would then have a graph where T_1 has an edge to T_2, and there is a subgraph of S, which is not connected.

Algorithm 13 creates the DAG for supporting the execution in the fallback path. The algorithm utilizes a concurrent union-find data structure [46] for each transaction and merges them, with the lowest union-find ID being the parent. By doing that, we create our order of transactions from child to the earliest conflicting transaction in the DAG.

Upon performing the necessary unions of all transactions, we have clustered elements into sets based on whether there are conflicts. In parallel, we also transform the DAG into a matrix representation, in which:

- $D_{i,j}$ is zero if i does not need to occur before j in the originating block.
- $D_{i,j}$ is 1 if i must occur before j in the originating block.
- $D_{i,i}$ is 1 if i is the first transaction within a cluster.

Algorithm 13 DAG Creation.

```
Require: txs is the set of transactions
 1: N = size(txs)
 2: uf = create_union_find(0...N-1)
 3: parallel for j = 0; j < N; ++j do
 4:     parallel for i=j; i < N; ++i do
 5:         if txs[i].conflicts(txs[j]) then
 6:             uf[i].union(uf[j])
 7:         end if
 8:     end parallel for
 9: end parallel for
10: D = upper_triangular_matrix(N, N)                ▷ All zero matrix
11: parallel for i = 0; j < N; ++j do
12:     idx = uf[i].find().id()
13:     D_{idx,i} = 1
14: end parallel for
```

In general, if $D_{i,j}$ and $D_{i,k}$ are 1 and $j < k$ then running i, j, k in this order corresponds to the ordering within the block. As it will be clear in Section 4.6, this matrix representation is very effective in moving between the CPU and GPU and keeping locality when traversing the matrix.

The fallback path is given the set of transactions, the just constructed DAG, and the set of transactions aborted in the fast path. The algorithm unfolds as follows.

- Transactions aborted in the fast path are rolled back through compensating transactions. In addition to that, all their dependent transactions, which were committed in the fast path, are also rolled back (i.e., compensated). Understanding transaction's dependencies is done deterministically using the DAG (Algorithm 14). At this point, all incorrect changes to the shared state have been annulled.
- All subgraphs of the DAG are executed in parallel, while transactions within a subgraph are run serially. At this stage, transactions' writes are buffered to simplify the

Algorithm 14 Executing DAG of Transactions.

Require: txs is transactions, D is the DAG, aborted_fastpath is set of transaction aborted in the fast path

```
1: N = size(txs)
2: parallel for i = 0; i < N; ++i do
3:     if D_{i,i} = 1 then
4:         redo = false
5:         for j = i; j < N; ++j do
6:             if D_{i,j} = 1 and j in aborted_fastpath then
7:                 redo = true
8:             end if
9:         end for
10:        for j = N - 1; redo and j ≥ i; –j do
11:            if D_{i,j} = 1 then
12:                txs[j].compensating_transaction()
13:            end if
14:        end for
15:        for j = i; redo and j < N; ++j do
16:            if D_{i,j} = 1 then
17:                txs[j].run() is aborted
18:            end if
19:        end for
20:    end if
21: end parallel for
```

abort procedure and skip the need for compensation. Still, the result of read operations is not recorded because the DAG already provides information about transactions' dependencies.

- If a transaction in a subgraph aborts (i.e., insufficient funds), its modifications are simply not applied to the shared state.

4.5.2 Schedule Equivalence

The state of the blockchain after executing a block with OranguTAN is equivalent to the state achieved if the transactions in the block were to be executed sequentially. Also, this schedule is deterministic. We now discuss this equivalence.

First, we consider a set of transactions with dependencies when proving that OranguTAN is equivalent to the sequential execution of the block. To determine correctness, we must consider two cases: all transactions successfully withdrawn funds, and at least one

transaction aborts.

If all transactions can successfully perform withdrawals initially in the fast path, then there is enough currency for an account to perform its transactions if re-ordered with any other transaction. By withdrawing all currency in parallel in the fast path before depositing it to complete the send-receive transactions, OranguTAN effectively sets aside the currency for each transaction to complete in parallel.

If one or more transactions abort, the DAG provides a schedule that respects the order within the block. If two transactions have dependencies, either the two transactions share a sending or receiving account, or there is a transitive dependency between them (i.e., T_1 depends on T_2, which depends on T_3). This is detected through the use of the union-find data structure. The built graph preserves the order of transactions in the block. By executing this subgraph in sequence (i.e., compensating and re-executing transactions), the order of the block is respected.

If we consider a set of dependent transactions and a transaction that does not conflict with any of the transactions in the set, ordering the independent transaction in any way will result in an equivalent state. If we consider more than one transaction that does not conflict with the set of dependent transactions, we can determine inductively that either this new set of transactions can interleave (i.e., do not conflict) or are dependent but will similarly respect the order of the block.

OranguTAN execution is equivalent to the sequential execution of the block because sets of dependent transactions respect block order, and sets of independent transactions can be reordered arbitrarily and still respect block order.

4.6 Exploiting Heterogeneity

The concurrency control illustrated in Section 4.5 has been designed to take advantage of the heterogeneous, highly parallel hardware of graphics processing units (GPU), often available on blockchain nodes. In fact, even low-end GPUs (commodity) are significantly more parallel than CPUs, and therefore, effectively exploiting both can achieve high and cost-effective performance.

OranguTAN needs to efficiently execute the fast path, create the DAG, and execute the fallback path. Our implementation has a hash map resident on the CPU that serves as the data repository (storing key-value pairs associated with users' accounts). To exploit locality and consider the affinity of transactional code with the CPU, executing transaction processing (fast path and fallback path) on the CPU while creating the DAG on the GPU is appropriate. By offloading DAG creation to the GPU, we can parallelize the memory movement (to and from the GPU) and the DAG creation with the transaction execution on the fast path. This enables lower latency when the fallback path is utilized since the GPU will start creating the DAG as soon as a block begins to be processed.

The DAG creation algorithm is more suitable for GPU due to the GPU's inherent parallel processing capabilities. Although the use of the atomic operations, which are required to implement the union-find data structure, are known to cause contention on the GPU [129, 68], we do not expect many transactions to conflict. In our experience, even with a workload with moderate contention, the high degree of parallelism still improves performance over creating the DAG on the CPU.

The GPU provides us with multiple levels of memory and the ability to utilize many threads to create the DAG. As detailed below, we take advantage of tiling with thread blocks (i.e., a collection of threads assigned to a physical GPU streaming multiprocessor). Within a thread block, we can have as many as 1024 threads. These threads can utilize global memory (i.e., the memory attached to the GPU) but can also utilize a very fast byte-addressable cache called shared memory.

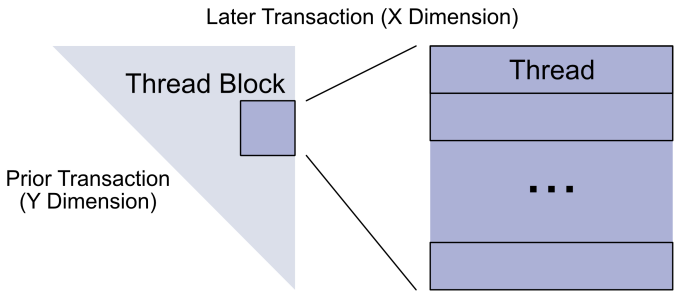


Figure 4.2: Tiling of DAG Union-Find Creation Step on GPU.

Two kernels are offloaded to the GPU: one for creating the DAG and one for representing

Algorithm 15 DAG Union-Find Step on GPU.

Require: txs is the set of transactions; uf is the union-find data structure; shared is a shared memory array

```
1:  N = size(txs)
2:  Blocks = N / threadsPerTile
3:  parallel for i = 0; i < N; ++i do
4:      parallel for block = 0; block < Blocks; ++block do
5:          start = block * threadsPerTile
6:          end = (block + 1) * threadsPerTile
7:          j = start + threadIdxInBlock
8:          shared[threadsPerTile] = txs[j]
9:          tx = txs[i]                                  ▷ Stored in register file
10:         synchThreadBlock();
11:         for j = start; j < end; ++j do
12:             if i ¡ j and tx.conflicts(shared[j - start]) then
13:                 uf[i].union(uf[j])
14:             end if
15:         end for
16:     end parallel for
17: end parallel for
```

it as a row-major upper triangular matrix.

In order to optimize the DAG creation algorithm for the GPU, we utilize the tiled-based approach seen in Figure 4.2. We loop over pairs of transactions. The looping resembles the same looping that may occur over every element of an upper triangular matrix. In this matrix, rows and columns are transactions. Each row is assigned to one thread and carries the conflict identification for one transaction. Therefore, its access set is stored on the register file for fast access. Conflicts are identified by looking at the row columns corresponding to transactions ordered after in the originating block. Because the transaction access sets are small in our SRM programming model, they all fit into shared memory, and therefore GPU threads can access them with low latency during the comparison for conflict identification. Algorithm 15 shows the pseudocode of this GPU kernel.

The union-find implementation relies on performing compare and swaps (CAS) to minimize the path to the root node of the set. We find the GPU can execute these operations efficiently for the type of workloads we evaluate in Section 4.7.

Another kernel creates a row-major upper triangular matrix representing this DAG. In this kernel, we assign each thread to a set of transactions to handle and perform the

operations. We find that, in practice, the most overhead occurs during the conflict detection kernel.

Once these kernels have been executed, we can copy the upper triangular matrix that represents the DAG back to the CPU. Using the upper triangular matrix instead of copying the DAG, we can perform the copy with a single memory copy operation. Also, since the matrix is row-major, traversing it to find conflicting transactions (i.e., rows where $D_{i,i} = 1$ and $D_{i,j} = 1$ and $i < j$) has good spatial locality.

4.7 Evaluation

We evaluate OranguTAN against state-of-the-art deterministic transaction processing concurrency controls suitable for blockchain. Specifically, we compare against Strife [149], Block-STM [71], and a version of OranguTAN where we only utilize our DAG-based concurrency control without the fast path. For simplicity, we call this implementation Dag. Block-STM utilizes pre-commit and commit phases to parallelize execution without knowing which transactions may conflict. Strife clusters transactions based on their read and write sets to process them deterministically. Our Dag computes the DAG to be used and only needs to execute the transactions once, rather than using it as a fallback path with compensating transactions like in OranguTAN. Unless otherwise mentioned, Dag also utilizes GPU acceleration like OranguTAN. We also compare against the concurrency control used in SPEEDEX [153], an approach that only allows for commutative transactions to be executed, and it does so by utilizing semaphores[3] to protect each currency unit moved.

To evaluate the performance of each concurrency control, we utilize SmallCoin, as presented in Algorithm 5, and SmallCoin+, as in Algorithm 8. Unless otherwise mentioned, we utilize uniform distributions to determine the sender and receiver accounts. We notice in practice that these uniform distributions often have minimal conflicts and are unlikely to cause aborts. We also utilize Zipfian distributions to increase the level of contention. As a result, using a uniform distribution demonstrates OranguTAN's ability to utilize the fast path, while using a Zipfian distribution solicits OranguTAN's fallback path.

[3]In Section K.6 of [153], the concurrency control "must lock the number of units of assets that could be debited" by utilizing hardware atomics. This is equivalent to acquiring semaphore.

We implement each of the concurrency controls in C++ and utilize CUDA 12.4 and GCC 11 to build it. When testing each workload, we generate 100 blocks of 1000 transactions beforehand. Then, we execute the concurrency control on the blocks to keep the system constantly loaded with work. We report the throughput for executing each transaction and the latency of executing the block.

We measure performance on an Oracle VM.GPU.A10.2 with a 30-core 2.6 GHz Intel Xeon Platinum 8358, hyperthreading enabled, and two NVIDIA Tesla A10s. We only utilize one GPU. This enables us to measure performance with 30 cores and 60 hardware threads. However, since public blockchains' nodes are not only limited to enterprise machines, and consumers may also participate with less expensive hardware, we study performance using an alternative, relatively inexpensive testbed in Section 4.7.1. Specifically, we compare performance on a consumer-level machine with a 6-core Intel i5-10600 (hyperthreading enabled) and an NVIDIA GTX 1660 SUPER. When evaluating a consumer-level machine, we observe trends similar to those of enterprise machines.

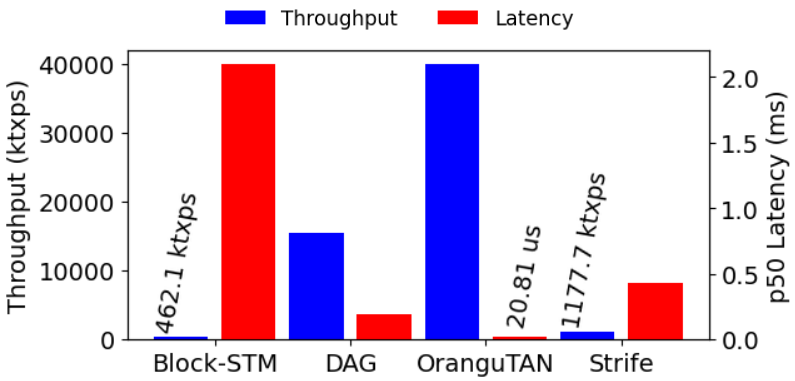


Figure 4.3: Uniform distribution; 1M accounts; SmallCoin.

We first compare the performance with the concurrency controls that can handle non-commutative transactions, namely OranguTAN, Dag, Strife, and Block-STM. Results are shown in Figure 4.3 when we use 1M accounts. We omit a comparison with the sequential execution of the transactions in Figure 4.3; however, when comparing, we observe that at 1M accounts, sequential can achieve higher throughput and lower p50 (50th percentile) latency than Strife and Block-STM. This suggests that the overhead presented in Strife of clustering transactions into non-conflicting sets and the overhead of the Block-STMs pre-

commits are significant. Sequential can perform well because of the high locality of the CPU cache's accesses to the map. Conversely, Dag can benefit from GPU acceleration and limited conflicts on shared cache lines due to how non-conflicting transactions are partitioned among threads. OranguTAN outperforms all other concurrency controls because of its ability to exploit commutativity. Even if there are conflicts between some transactions, the conflicts consist of a few atomic operations. The parallelization helps OranguTAN achieve a 2.6x improvement over Dag and a 3.9x improvement over executing the block sequentially.

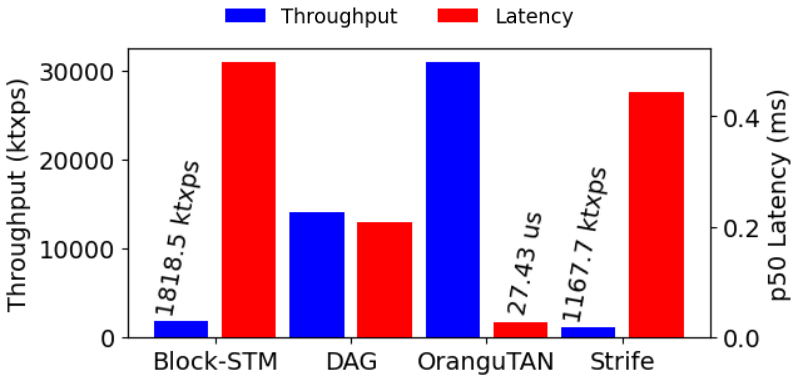


Figure 4.4: Uniform distribution; 1M accounts; SmallCoin+.

A similar trend can be seen when looking at SmallCoin+ in Figure 4.4. OranguTAN outperforms Dag while Strife and Block-STM have limited performance due to the respective overheads of the concurrency controls. On both the SmallCoin+ and SmallCoin workloads, OranguTAN achieves 2.2x and 2.6x the throughput of Dag. This suggests a consistent performance gain when using OranguTAN compared to Dag thanks to the fast path. Both Dag and OranguTAN similarly drop in some performance when moving from SmallCoin to SmallCoin+ due to the increase in transactional operations.

OranguTAN retains its performance advantage over Block-STM when moving from SmallCoin to SmallCoin+. In SmallCoin, OranguTAN is 86.4x the throughput of Block-STM. On SmallCoin+, OranguTAN has 17x the throughput of Block-STM. The throughput of Block-STM also increases from 462 ktxps to 1819 ktxps on SmallCoin+. This is due to the optimistic design of Block-STM. While each transaction's read and write set stays the same, the transactions must perform more operations. This effectively slows transactions, lowering contention in the system and limiting the amount of transactions that must be

redone.

OranguTAN's performance gain lessens over Strife when moving from SmallCoin to SmallCoin+. In SmallCoin, OranguTAN is 33.9x the throughput of Strife, but is 26.5x the throughput in SmallCoin+. The throughput of Strife is relatively similar from SmallCoin to SmallCoin+, 1178 ktxps to 1168 ktxps, respectively. That suggests that Strife can better amortize the cost of the clustering, as the transactions require more time to execute due to the increase in operations.

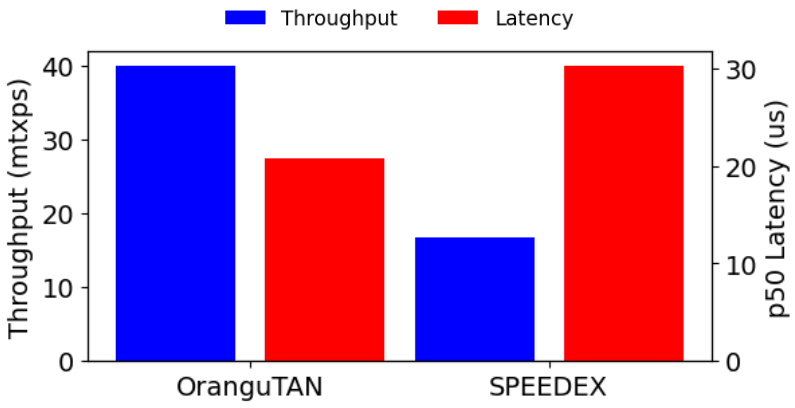


Figure 4.5: Uniform distribution; 1M accounts; SmallCoin; Commutative Operations.

When we compare the performance of OranguTAN to the SPEEDEX approach in the case that all transactions commute, we find that OranguTAN can also outperform SPEEDEX by 2.4x. Recall that SPEEDEX does not need to instrument transactions since their operations are commutative by construction. In Figure 4.5, we observe that OranguTAN has a higher throughput and 31.1% lower latency than SPEEDEX. Although both approaches use few atomic operations, SPEEDEX needs to atomically update the shared semaphores associated with an account before processing the transaction. For example, if SPEEDEX sends 10 credits from account A, it must acquire 10 credits from A's semaphore. Once this is done, it can get and update A's value. OranguTAN, instead, will perform a single atomic operation on A's value. This means that in the ideal case, OranguTAN executes two read-modify-write atomic operations for a SmallCoin transaction (withdraw from A and deposit to B) while SPEEDEX must execute three (acquire semaphore on A, withdraw from A, deposit to B).

Similar trends emerge when observing the performance of OranguTAN versus SPEEDEX

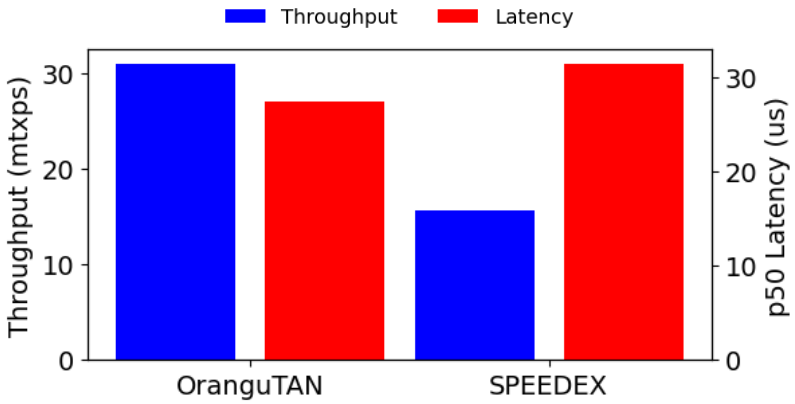


Figure 4.6: Uniform distribution; 1M accounts; SmallCoin+; Commutative Operations.

in Figure 4.6. OranguTAN has 2x the throughput of SPEEDEX and a 12.6% lower p50 latency. This suggests that OranguTAN and SPEEDEX similarly degrade in performance while increasing the number of reads and writes that must occur within a transaction, but the use of semaphores versus atomic operations leads to the lesser performance of the SPEEDEX approach.

Table 4.2: Dag - CPU vs GPU Performance.

Algorithm	Throughput	p50 Latency
CPU Only	2.5 mtxps	405 us
GPU Accelerated	14.2 mtxps	206 us

Next, we look at the tradeoffs of using the CPU versus the GPU for the DAG creation. We evaluate how the Dag concurrency control performs when using only the CPU versus both CPU and GPU in Table 4.2. When using the GPU, it can handle 5.8x the transactional throughput and reduce p50 latency by 49.2%. This might look surprising, considering the overhead of communicating with the GPU and offloading kernel execution. However, two reasons justify our speedup. First, the DAG creation using our technique is significantly faster on the GPU than on the CPU. Second, the ability to pipeline the GPU execution with the CPU execution allows for further increases in throughput.

We also evaluate competitors by looking at the fallback case and varying the skew of the access distribution. We use two Zipfian distributions to choose the sender and receiver accounts. By increasing the θ parameter of the Zipfian, which affects the likelihood of transactions accessing the same accounts, we also increase the number of transactions that

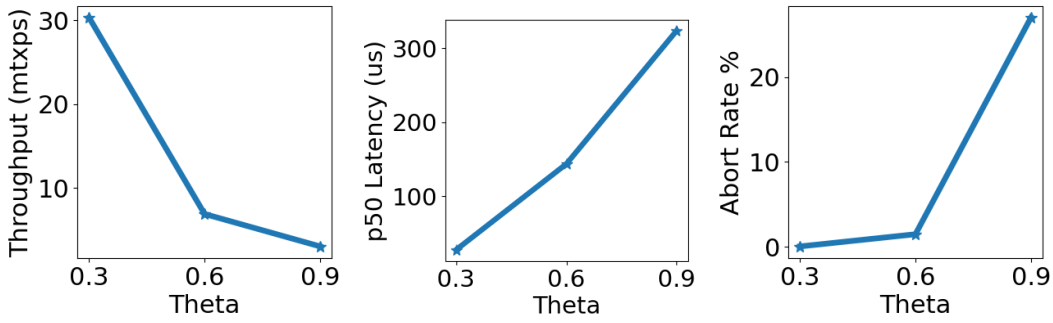
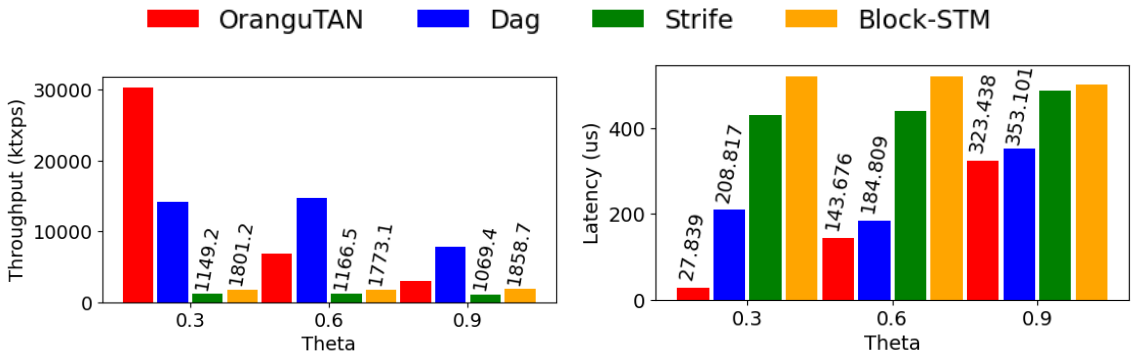


Figure 4.7: OranguTAN with Non-Commutative Transactions.



(a) Throughput

(b) p50 Latency

Figure 4.8: Zipfian Distribution; 1M accounts; SmallCoin+.

cannot commute (see the increasing abort rate in Figure 4.7). Transactions begin using the fallback path at a θ of 0.4. As we increase θ from here, the abort rate increases up to 27% at $\theta = 0.9$. As θ increases, throughput decreases from 30M tx/s to 2.9M tx/s. The p50 latency of executing a block increases from 28 us to 323 us.

Increasing contention implies a degradation of OranguTAN's performance. Figure 4.8 shows results using competitors that support non-commutative transactions and by varying θ. OranguTAN still outperforms them. At θ=0.6, the throughput of the GPU-accelerated version of Dag is 2.2x the throughput of OranguTAN. This gap increases until Dag has 2.6x the throughput of OranguTAN. However, OranguTAN has a lower p50 latency than Dag. This suggests that the use of commutativity can lower the latency of processing blocks of transactions. However, because the Dag can more efficiently pipeline between the CPU transactional execution and the DAG creation on the GPU, it can achieve a higher throughput when there is high contention.

Compared to Strife and Block-STM, OranguTAN is able to achieve consistent performance improvements. The improvement in throughput over Strife ranges from 2.8x to 26.4x. This improvement is due to the ability to commute transactions and rely on a GPU-accelerated fallback path rather than just utilizing an approach that clusters transactions. Compared to Block-STM, OranguTAN can achieve a 1.6x improvement in performance to 16.9x. This is due to OranguTAN not needing to pre-commit and resolve dependencies like Block-STM. If a transaction aborts in OranguTAN, a simple resolution in the fallback path occurs. OranguTAN also significantly outperforms both in terms of its p50 latency.

4.7.1 Consumer-Level Hardware

We also evaluate the performance of all competitors on a consumer-level machine. In Figure 4.9, we run the same Smallcoin uniform distribution configuration as in Figures 4.3 and 4.5. Similarly, we find OranguTAN achieves greater throughput and lower latency than competitors with 19.4 mtxps and a 0.04 ms latency. However, in this machine configuration, OranguTAN has a 2.7x speedup over Dag compared to the 2.6x speedup over Dag on the enterprise machine. This is expected because the enterprise testbed has 6.5x the GPU cores and 5x the CPU cores of the consumer machine (1408 compared to 9216 and 60 compared to 12, respectively), meaning OranguTAN effectively scales in both CPU and GPU performance from consumer to enterprise hardware.

A similar trend is found when considering SmallCoin+ in Figure 4.10. OranguTAN is 2.5x faster than Dag, which is greater than the 2.2x speedup OranguTAN achieves over Dag for the enterprise-level machine.

In comparison to Block-STM, Strife, and SPEEDEX in Figures 4.9 and 4.10, OranguTAN still maintains its improved performance due to its ability to commute operations. OranguTAN achieves at least 15.1x the throughput on Smallcoin and 7x the throughput on SmallCoin+ over non semantic-aware approaches. Compared to SPEEDEX, when OranguTAN can commute operations, it only requires two read-modify-write operations with respect to SPEEDEX's three (as noted above). This is still an advantage on the consumer-level hardware, leading to a 1.8x and 1.5x speedup on Smallcoin and Smallcoin+, respectively. OranguTAN acheives a 32.2% reduction in latency on Smallcoin and a 19.6% reduction in

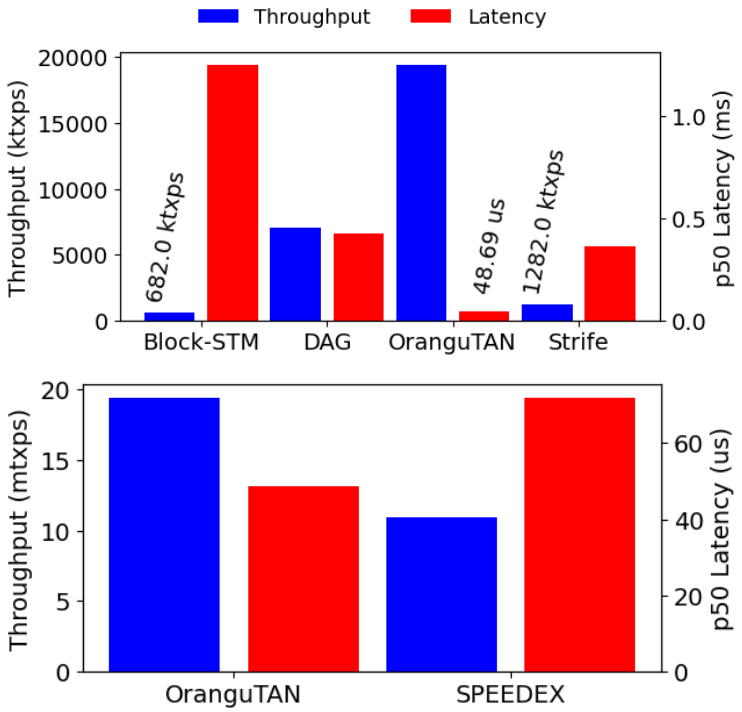


Figure 4.9: Uniform Distribution; 1M accounts; SmallCoin; Consumer Machine.

latency on SmallCoin+.

Table 4.3: Dag - CPU vs GPU Performance on Consumer Hardware.

Algorithm	Throughput	p50 Latency
CPU Only	773 ktxps	1.3 ms
GPU Accelerated	6264 ktxps	0.5 ms

We also compare the performance of CPU only versus GPU accelerated Dag on consumer hardware in Table 4.3. Using GPU acceleration leads to a 8.1x speedup. We only observe a 5.8x performance improvement on the enterprise machine. This is because the CPU-only Dag cannot parallelize as much with only 12 hardware threads. On consumer machines, this suggests that utilizing an inexpensive GPU to parallelize parts of a blockchain concurrency control is an effective strategy to improve performance significantly.

4.8 Related Work

Blockchain systems have innovated in the ability to run secure distributed transactional processing. Both industry and academia have shown vast interest in the last decade. A

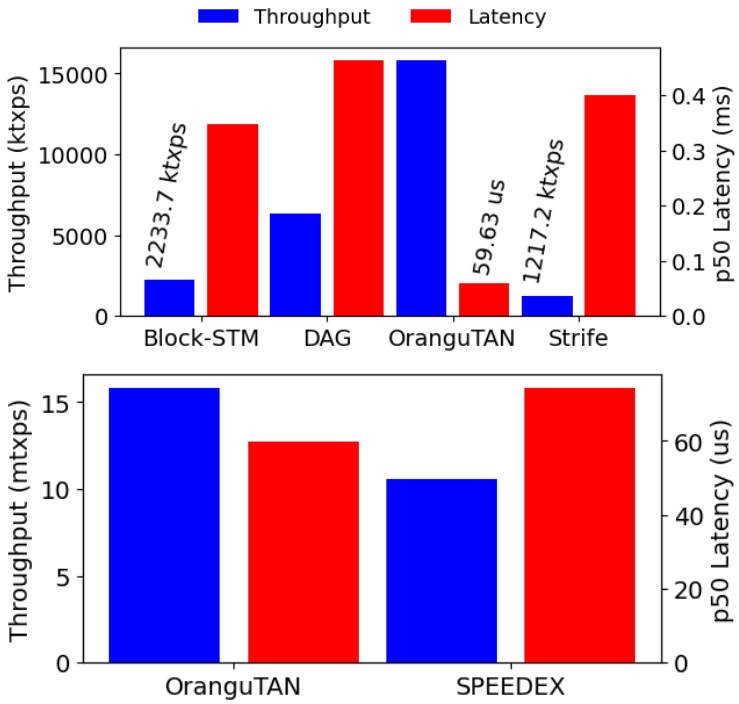


Figure 4.10: Uniform Distribution; 1M accounts; SmallCoin+; Consumer Machine.

variety of blockchains, including Aptos [66], Ethereum [185], Stellar [170], and others, can be utilized for simple payment systems with a model equivalent to our send-receive-money programming model. Recent innovations have focused on utilizing these operations to create more complex transactions through smart contracts (e.g., Algorand's smart contracts [9]). This chapter focuses on the send-receive-money transactional model and the deterministic concurrency controls that process these transactions.

Traditional deterministic transactional processing approaches, such as Calvin [175], are designed for conflicting distributed transactional workloads where requests to run transactions are streamed to the system. In a blockchain, these transactions enter the system as a block of transactions that must be executed together.

Related to our work is the Strife [149] concurrency control, which processes collections of transactions by clustering them based on their read and write sets. Similar to Strife, OranguTAN groups transactions into collections that can be parallelized. OranguTAN and Strife use the union-find data structure [46] to parallelize the procedure that groups transactions. Unlike Strife, we do not use a heuristic approach; instead, we consider all potential

conflicts between all transactions and utilize a heterogeneous architecture to support that efficiently.

Block-STM concurrency control [71] is designed to be utilized by the Aptos blockchain. Block-STM utilizes multi-versioning to pre-commit transactions within blocks and detect conflicts between transactions. When a transaction aborts due to conflicts, it can be rolled back and re-executed, and any conflicts can be appropriately handled through tracking the multi-versioning. OranguTAN handles aborts of conflicting transactions through roll-back, re-execution, and handling of conflicts through a directed acyclic graph (DAG) of transactions. Unlike Block-STM, OranguTAN detects conflicts by exploiting transactional semantics and uses a deterministic procedure with a DAG of transactions.

Block-STM has been further extended by introducing RapidLane [123]. RapidLane enables deferring computation till commit time for commutativity. The programming model introduced enables speculating whether operations will be successful (e.g., incrementing a global counter), and then, when no prior transactions in the block are re-executed, the computation can be completed. Unlike RapidLane, OranguTAN allows the parallel execution of conflicting (read/write) transactions by converting transactional operations into atomic operations. OranguTAN also avoids speculation by restricting the programming model, while RapidLane allows for a more general programming model (e.g., including smart contracts).

SPEEDEX [153] is a recent work on blockchain for decentralized exchange. Similar to OranguTAN, it also focuses on commutativity. SPEEDEX utilizes semaphores to semantically acquire the amount of currency needed to run the entire set of transactions (e.g., for two transactions which each send 10 credits from account A to B, SPEEDEX will acquire a semaphore 20 times before executing the set of transactions). This enables SPEEDEX to enforce the commutativity of all transactions. However, this prevents SPEEDEX from handling a set of non-commutative transactions. OranguTAN is designed to handle blocks with transactions that do not commute while benefitting from the commutativity. OranguTAN also uses atomic operations rather than semaphores to enable transactions that commute to execute efficiently.

Lastly, Groundhog [154] extends the ideas presented in SPEEDEX to a more general programming model. More specifically, it utilizes the same concept of reserving currency

through acquiring a semaphore. While OranguTAN does not focus on a general programming model, we believe that OranguTAN's utilization of splitting transactions into two parts could similarly extend to a more general model.

4.9 Summary

In this chapter, we presented OranguTAN, a deterministic transaction processing scheme that can be used by blockchain proposer and validator nodes when dealing with transactions that transfer money between accounts. OranguTAN aggressively exploits the commutativity property of these simple transactions to eliminate the need for instrumentation at runtime. As transactions are processed, the GPU builds a dependency graph in parallel. This graph is subsequently used by those transactions that cannot commute. OranguTAN's evaluation study demonstrates that blockchain transaction processing can be significantly sped up by taking advantage of the simple transactional semantics.

Chapter 5

Generalizing the Coalescence Optimization Across Architectures

5.1 Introduction

Data structures have been widely studied within the context of CPUs, especially in employing SIMD instructions to optimize memory access patterns [157, 147, 163], but emerging architecture such as GPUs and remote direct memory access (RDMA) challenge how we design data structures to utilize our hardware fully.

GPU data structures have been utilized in a variety of applications, from data preprocessing in cuDF [173], caching embeddings for recommender systems in Merlin [144], to storing and processing data in RateupDB [107]. Along with many publications that study or use GPU data structures [192, 53, 14, 22, 21, 23, 18].

RDMA data structures are utilized in distributed systems for optimal performance that extends beyond the memory of the local machine [181, 164, 182, 58, 131]. The memory locations can be disaggregated, as well as the location of the processors. With the emergence of CXL, data structures for these environments will only become more important [183, 79, 180].

Both GPU and RDMA-based programming models present coherent access to memory through a shared-memory model. The latency of accessing this memory is significantly high,

and operations should occur within the streaming multiprocessor of a GPU or the local node of an RDMA cluster to the best of a programmer's ability. However, memory accesses are always necessary for data structures. A trivial port of many CPU data structures would lead to low memory bandwidth utilization and unexpectedly higher latency operations, causing poor performance, especially for higher thread counts in GPUs and RDMA clusters.

To achieve optimal performance when accessing this memory, it is best to make use of the high bandwidth it provides (4.8 TB/s HMB3e on an NVIDIA H200 GPU [143] and 800 Gb/s on a Connect X-8 Infiniband network card [140]) by optimizing for *coalescence*.

Coalescence is a software optimization designed to minimize the number of memory transactions by accessing more than one memory object at a time (e.g., if we have a program that will read in an array from memory containing 2 elements and will sum them, it could read in the entire array in a single load to minimize memory transactions). This optimization has been traditionally used in data structures such as B-trees, where multi-element nodes are utilized to minimize the number of operations we must do on our hard disk. However, it is emerging as a software optimization in seemingly unconnected modern architectures.

In this work, we present COalesced Data Structures, CODS, a programming model of coalescence for use in designing data structures for CPUs, GPUs, and RDMA. CODS enables designing generalizable approaches to coalesced data structures by using common APIs that can be implemented for each architecture. The authors are unaware of another work that has generalized coalescence across these architectures.

We enable a variety of methods in CODS to map coalescence to hardware, enabling threads on GPUs to coalesce accesses from DRAM to registers, CPUs to utilize SIMD instructions to coalesce accesses, and RDMA to copy large chunks of memory between machines without remote CPU involvement. When tuned correctly, coalescence can improve the throughput of simple data structures like CPU-based linked lists by at least 42% depending on the workload and can increase performance by 2x or greater on RDMA and CPU trees. Through the CODS programming model, we can make these trade-offs in coalescence per operation on the GPU to maintain 2% to 10% gains in mixed read and update workloads and up to 3x on read-only workloads.

The CODS programming model and corresponding C++ implementation can enable us

to reproduce state-of-the-art data structures that already employ coalescence optimizations. We also demonstrate that CODS aids in designing portable data structures for various architectures. Furthermore, with generic CODS-based data structures, programmers can optimize how they coalesce based on the workload and desirable performance characteristics. We demonstrate that this programming model maps well to CPU, GPU, and RDMA based systems, however we believe that this pattern can be even further generalizable beyond these three architectures.

In Section 5.2, we overview the CODS model as well as how a lazy linked list based chaining hash set with coalescence optimizations maps to CPUs, GPUs, and RDMA, and give a general idea of how this would map to the CODS APIs. We then discuss the programming model of CODS in more detail in Section 5.3 and how it maps to each architecture and their forms of coalescence. In Section 5.4, we discuss how to develop a lazy linked list in CODS as an example of the power of this model. We use the two varieties of atomicity CODS provides: full coalescable atomicity, where each coalesced operation is atomic, and elementwise atomicity, where coalesced operations only guarantee elementwise atomicity. Finally, in Section 5.5, we evaluate CODS.

5.2 Overview

In CODS, we present a unified model for designing and reasoning about coalesced data structures. Within this model, we have coalesced objects, c-objects, which contain a group of elements that we wish to access in a coalesced manner. In the most basic sense, a c-object is an array with metadata describing alignment and whether elements can be accessed in a vectorized fashion. A special pointer to these c-objects, which we call a c-object pointer, provides the ability to coalesce accesses and manipulate individual elements. When a programmer wishes to operate on the data in the c-object, they can utilize the pointer and its associated APIs, along with arguments on how to coalesce accesses. CODS presents the following operations to programmers:

- *Atomic load* will be given an index of an element as an argument and will atomically load that element and return the value.

- *Atomic store* will be given an index of an element as an argument and will atomically write to that element.
- *Get* will be given the method of coalescence and will read in the c-object in a coalesced manner.
- *Set* will be given the coalescence method and will store a c-object out to memory in a coalesced manner.
- Other operations, such as compare and swap, are possible.

The coalescence method, which can depend on the underlying hardware and software runtime, and guarantees about atomicity are of utmost importance when using this model to design data structures. The primary coalescence method guarantees elementwise atomicity when coalescing (i.e., if two elements A and B are coalesced, the access of A and B can occur in any order, and other operations by other threads can interleave between A and B). Still, CODS also enables a method to make all coalesced accesses atomic across all elements (i.e., when coalescing access to two elements, A and B, another thread can't interleave its operation between the access of A and B). We can generalize data structures and techniques between domains with our CODS model. Furthermore, we present a variety of high-performance data structures for this model.

We will begin with an example of how a simple unoptimized chaining hash set may be designed in CPU, RDMA, and GPU environments to demonstrate the commonality in these environments that supports insert and contains. The chaining hash set will be built from a lazy linked list for each bucket. After explaining how this data structure may be coalesced in each environment, we will explain how CODS enables a more general approach to this problem.

5.2.1 Motivating Example: Chaining Hash Set

Consider a CPU-based chaining hash set; a simplistic implementation of this data structure could utilize an array of concurrent linked lists with the well-known lazy linked list technique [87]. If we want to employ coalescence for this data structure, we may increase the

size of our nodes to contain up to 2 elements. When searching for our keys, we can either utilize SIMD instructions or benefit from the temporal-spatial locality of the accesses to the node.

The logic becomes more complicated when performing insertions than when using our traditional single-element lazy linked list. After searching for our key and locking the previous and current nodes, we must reread them to validate that the previous node does not contain our key or have elements less than our key. If we do not traverse to the end of the list, we must also read the current node to validate that it does not contain our key.

To maintain order in a coalesced lazy linked list, we must implement splitting, taking a locked node, and partitioning by the key we intend to insert into two nodes. This must be done by creating a copy of our current node and a new node, and we can then partition by our key. The copy must be made to ensure readers cannot observe an inconsistent state. Once done partitioning, we can insert our key into one of the nodes and atomically link in our nodes. While adding some complexity to the implementation, we may now benefit from the SIMD instructions and temporal-spatial locality. If we design this generically for our CPU, it can be portable to other SIMD instructions (e.g., various SIMD sizes and implementations).

Modern GPU single instruction multiple thread (SIMT) architectures also enable coalescence. When a group of 32 threads, called a *warp*, executes the same instruction and accesses memory within the same cache line, the memory transactions get consolidated into a single memory transaction. We can reduce the effective number of memory transactions through coalescence and improve system performance. If we wish to implement a chaining hash map design on the GPU, we may choose to pursue an approach similar to SlabHash [18] or L-Slab [53], where we utilize a linked list where each node is 32 elements wide. We can design with a warp cooperative approach when programming our data structure. In a warp cooperative approach, we utilize a warp rather than a single thread to perform an operation. For example, when we wish to read a node, we can parallelize over threads to read the node in a single coalesced transaction.

If we were to implement our ordered lazy linked list for the GPU, we would have a lock per 32-element node. We can utilize a similar technique to the one presented for

L-Slab [53] to have warps collectively lock nodes. We perform a coalesced read of each key to our register file when we read a node. It is worthwhile to note that the memory model presented in the NVIDIA PTX documents does not provide any guarantees about the ordering of operations for coalesced transactions (i.e., we cannot guarantee that if a warp coalesces memory accesses it appears as a single atomic operation, it may appear as a collection of smaller atomic operations) this is further complicated by the memory model [113, 8]. However, since we will lock when inserting or removing, we can always re-validate when we lock a node to confirm semantic changes have not occurred, similar to on the CPU. Our node splits, necessary for insertions, can now be written with GPU intrinsics to perform parallel partitions of nodes. This coalescence optimization is done in [18] and [53] to enable better scalability on GPU hash maps.

Remote direct memory access (RDMA) based systems, such as modern Infiniband-based disaggregated memory systems, enable coalescence. In these systems, multiple computers access remote memory through the network. The programming model presents traditional two-sided APIs like send and receive and one-sided APIs like read, write, and CAS. In both APIs, the network card can bypass the kernel and directly read and write to memory. In two-sided APIs, two nodes must communicate with both CPUs and be involved in the process; one machine must utilize the receive API while the other utilizes the send API for a successful interaction. However, one-sided APIs can be initiated from a single node to read, write, or atomically operate on the remote node's memory without the involvement of the remote node and copy the result back to the local node's memory.

Furthermore, one-sided APIs do not easily interoperate with memory operations issued by the CPU, creating atomicity issues. While the one-sided APIs can be limited, they provide a mechanism to develop high-performance shared memory systems. To achieve the best performance, a programmer should coalesce multiple elements in a single read or write to minimize the number of network round trips.

When implementing a chaining-based hash map for an RDMA-based system, we utilize spin locks through an RDMA CAS to lock linked list nodes for insertions or removal. To read a node, we can issue a single read of the entirety of the node to copy it to our local memory. Once copied, we can search the node for the relevant key. When we perform insertions or

removals, we can search for our key by copying each linked list node to our local memory. Once we find a node, we can lock, re-validate it, and perform our operation. To perform a split, we can allocate a node from our remotely accessible memory and partition locally. Then, we can perform an atomic write to update the global state.

In CPU, GPU, and RDMA-based coalescence, the typical memory hierarchy structure is at least two levels high. With CPU-based coalescence, we will take advantage of either temporal-spatial locality of accesses and expect to fetch data into the cache line. We may also utilize SIMD instructions to bring data into the register file. Effectively, we have a nearby cache or register file and a "far" DRAM where accesses will be much slower. With GPU-based coalescence, we will have a nearby register file and a far DRAM with slower accesses. In RDMA-based systems, local memory is near our processor and provides lower latency than the far remote memory. Our solution, CODS, abstracts this pattern and provides a model to the programmer to build data structures that require atomic and consistent building blocks to implement.

Within CODS, we can implement this linked list more generically. Each node consists of metadata (e.g., locks), which is not intended to be accessed in a coalesced manner, and a coalescable-object (c-object), which presents a way to store the data we wish to coalesce. In this case, we could have the keys as a c-object. The c-object is laid out in memory as an array of N elements. We will assume unsigned integer keys for simplicity, with the maximum representable integer being reserved as a null key. We will allocate and construct our node in the appropriate memory to use it. When we wish to access our node, we get a pointer object to the c-object; we call this pointer type a c-pointer. This c-pointer enables us to `get` a copy of the node, `set` a copy of the node, `set_subset` to set a subset of the c-object, `atomic_load` an element in the c-object, or `atomic_store` an element.

In our insertion algorithm, we can call `get` instead of SIMD intrinsics to get a copy of the node through a coalesced access. We can then work with this copy by accessing the elements within the copy. After validating, we can atomically append an element if the node is not full with `atomic_store`. Our partitioning for splitting full nodes can be done entirely in c-objects in our nearby memory and then store the result with a `set`. After this, we can unlock the node. These modifications make the data structure generalizable to CPUs,

GPUs, and RDMA. We can reduce the number of operations, reduce round-trips, reduce memory transactions, or increase cache hits depending on the architecture and mode of coalescence. This mode of coalescence is specified by the programmer when using `get` and `set` as well as through the configuration of the c-pointers and c-objects at compile time.

5.3 CODS Programming Model

CODS presents a general model for coalescence that can be adapted to model the coalescence of CPUs, GPUs, and RDMA. For any coalescence, we must have at least two levels of memory to work with. As mentioned in Section 5.2, we will call the low latency memory near and the high latency memory far. When we coalesce accesses between near and far memory, CODS will parameterize the accesses in a few ways to generalize between the memory copy-like API of reads and writes that RDMA presents, the single instruction multiple thread (SIMT) model that GPUs present, and the potential for GPUs or CPUs to vectorize accesses through SIMD instructions or multi-word accesses. When we perform a get or set with a pointer to a c-object, how the memory is loaded is parameterized by:

- The size of the elements in the c-object.
- The number of threads performing the load.
- How we vectorize loads and stores.

We can consider performing a get on a node of the lazy linked list presented in 5.2. When we perform this load with a GPU, we may perform with any number of threads intending to coalesce accesses together. Suppose we have N threads and M elements. In that case, the most straightforward way to coalesce accesses from DRAM to registers is to have each thread load M / N elements and have thread 0 load elements 0, thread 1 load element 1, and so on in the first load then thread 0 load element N, thread 1 load N + 1, and so on until each thread has loaded M / N elements. This means that in the register file, element K was loaded by thread K mod N, which is element K / N that the thread loaded. For example, with 32 threads and 64 elements, the 36th element will be loaded by thread 4, and will be the second element loaded by the thread. Once loaded,

we can utilize NVIDIA's thread group concept to share data between threads through its cooperative group programming model [84]. In this model, N threads can shuffle elements between each other, broadcast boolean values, and synchronize. We choose to implement CODS with this model of inter-thread GPU communication.

Now, consider performing the load with a vectorized instruction. If each load contains N elements, and we have M elements in total, then we must perform M / N loads. This means the Kth element maps to offet K mod N within load K / N. For example, if we have 16 elements and can load 4 elements per SIMD instruction, we can get element 5 in the second load at index 1 of the vector.

Putting both vectorization and SIMT models together, we denote elements either by their index in a c-object or their index with respect to the number of threads and the number of elements being vectorized. We denote this by either a single index or a thread, thread's vector, offset in vector index. For example, the ith element $E[i]$ with 32 threads and a vectorized load size of 4 corresponds to the i / 4 th vector that will be loaded overall. This vector will be loaded by thread i / 4 mod 32. For that thread, it will be the i / 128th vector. So the element is $E[i/4 \bmod 32, i/128, i \bmod 4]$. In both CPUs and RDMA, threads are always set to index 0. Furthermore, in RDMA CODS data structures, the offset within the vector is always 0 since there is no vectorization.

To set up a get or set to work with CPUs, GPUs, or RDMA, we will utilize the size of the elements in the c-object and determine the size of the largest element. Each element will be padded to this size (for example, a character and a 32-bit integer will be treated as two 32-bit elements). Next, the programmer can determine at compile time how many elements they wish to vectorize to. If supported, it will compile; otherwise, it will not. With a char and a 32-bit integer, we can vectorize up to a 64-bit load on GPUs and RDMA. We also pass an object to communicate the number of threads performing the load. We will only perform the load with a single thread if not provided. After this, we pass the method of coalescence.

We enable two methods of coalescence in CODS and a bypass that we enable the programmer to use:

- Fragment coalescence is a method to access a statically sized data fragment between near and far memory. This form of coalescence intends to enable compilers to transform the loads into loads to registers.

- Copy-pointer coalescence is a method to copy data between pointers in near and far memory. This method is utilized for RDMA, where the network card will perform the copy between a pointer in near and far memory.

- Bypass is a method that will perform a no-op for gets and enables accesses of each element from far memory, intending to be cached by the hardware in the memory hierarchy, such as in L2.

These forms of coalescence in CODS facilitate adaptation to a wide range of architectures and design decisions. For example, a CPU can either load with a vectorized copy into registers through fragment coalescence or use temporal-spatial locality by having CODS only load elements when accessed.

Once we have a copy from a get, we can perform *load* and *store* on indices. When passing a thread index, the vector, and the vector offset in a load, CODS will call for a shuffle between threads to broadcast the corresponding element. When not passing a thread index but passing a vector and offset into the vector, CODS will load or store the local thread's element at that vector and vector offset. This enables programmers to read and manipulate the copied data. Programmers can then either set or utilize atomic operations to modify the underlying c-object in far memory.

Since CODS is intended for use with data structures, it is also essential to discuss the guarantees of atomicity and the memory model that CODS provides. CODS always provides atomicity when accessing individual elements and can utilize the memory model of the underlying programming language to enable programmers to specify necessary memory ordering (e.g., sequential consistency or acquire-release semantics). For gets and sets, CODS does not always guarantee atomicity of the entire operation, only elementwise atomicity. For example, if thread 0 gets elements a c-object with two elements and thread 1 sets the c-object, thread 0 may observe an inconsistent state where thread 1 only partially sets the c-object. However, thread 0 will never see inconsistencies on a per-element basis. Ordering

these collections of operations is possible by inserting the correct fences and specifying the necessary memory ordering. In most cases, this is sufficient for designing high-performance coalesced data structures. When this is insufficient, CODS provides a method to specify that the gets and sets must be fully atomic.

CODS can support atomicity of the entire get and set if the c-objects are cache line aligned, and the following is true for each architecture:

- For a CPU, the get or set utilizes a single vectorized operation, and the vectorized operation is, at most, the size of a cache line.
- For RDMA, the get or set is only cache line size.
- For GPUs, the get or set only loads a cache line as a coalesced operation.[1]

If CODS provides fully atomic guarantees, we can utilize the underlying programming memory model to present the programmer with the ability to order gets, sets, and elementwise atomic operations. By utilizing a sequentially consistent memory order, we can implement a linearizable register with CODS that provides gets, sets, and atomic operations on subsets of the register. When CODS only provides elementwise atomic guarantees, the programmer can still order gets, sets, and elementwise atomic operations with respect to each other within their thread. However, operations may interleave across threads, and implementing a linearizable register larger than a single element is not possible.

We implement this model as a C++ library and build on the CUDA library to support GPUs [139], an RDMA-based framework called Remus [24], and the GCC SIMD intrinsics for x86_64. While the CUDA model is similar to traditional C++ and pointers are 64 bits, the RDMA framework must utilize a node ID, pointer, and key to access remote memory. We utilize known techniques and steal the most significant bits of the pointer to store the node ID for RDMA. When accessing a remote node, we utilize our key, which we must pass to every function. We encode this in our programs through a context object, which we pass around. As mentioned earlier, we must also pass around a thread group to synchronize accesses and coordinate between coalescing threads when utilizing GPUs.

[1] Although not guaranteed in the NVIDIA PTX memory model. The authors in [23] make this assumption.

In the following section, we detail how to adapt existing data structures and techniques to this model but will omit thread groups or RDMA contexts for simplicity. The novelty of the gets, sets, and per-element atomic operations, coupled with the atomicity and consistency, guarantees that we can represent and reason about coalesced data structures in these environments without being tied to a particular underlying software architecture or hardware architecture.

5.4 Data Structures with CODS

CODS enables us to implement various coalesced data structures and develop uniform design approaches. In this section, we discuss how to implement data structures with CODS in mind and how to use CODS to minimize the number of reads and writes needed. We focus on lazy linked lists as an example. These lists typically form the baseline of data structures, such as chaining hash sets or hash maps. The concepts we use in building coalesced linked lists can be further extended into data structures such as skip lists, such as skip vector [156], or B-trees, such as Awad et al.'s B-Trees [23, 21]. We evaluate general implementations of CODS-based B-trees in Section 5.5, which use similar concepts. Lazy linked lists have lock-free reads, which enables progress and is essential in scenarios with high contention, such as GPUs, or high latency, such as RDMA. We will first consider how to design these linked lists in fully atomic and then in elementwise atomic regimes.

Lazy linked lists [87] guarantee correctness through three primary design decisions: locking is used to order inserts and removes, nodes are marked deleted through bit-stealing or an internal boolean, modifications to the list are done solely by updating pointers, and the nodes are ordered according to values. Marking nodes as deleted enables coordinating between removes and other operations. By solely updating the list through updating pointers (i.e., inserting new nodes or removing nodes after marking them deleted), we can guarantee that conflicting operations will conflict on the updates of pointers. Ordering makes it possible only to scan a subset of the list. When enabling coalescence in lazy linked lists, the reasoning needed to guarantee correctness becomes more complex because modifications to the list are no longer solely done by updating pointers but may be done by mutating a

node's internals. In our CODS-based lazy linked list design, we wish to remain generalizable to RDMA, GPUs, and CPUs. For that reason, we are limited by 8B compare and exchange. This means we can only internally modify nodes of various sizes through gets, sets, and atomic operations on elements.

We can use a slightly modified search algorithm in fully atomic and elementwise atomic regimes for our coalescable lazy-linked list. Our c-objects will consist of elements and a bit set to denote whether elements have been deleted internally within the node. We can utilize standard bit-stealing techniques to mark our nodes as deleted and employ helping techniques with compare and swap to remove deleted nodes. When we traverse, we will use the `get` API to access the node and search through the returned copy. In the following insertion and removal algorithms, we will return a value denoting whether we reached the node, found an element greater than the node, or reached the end of the linked list and the previous and current node.

We can also guarantee the number of memory operations our search requires through CODS. If we assume that the nodes contain N elements, then in the absence of removals, if a list contains n elements, the search will require $\lceil n/N \rceil$ coalesced reads in the worst case. Depending on the maximum number of elements we can operate on in a single memory transaction (i.e., a 128B L1 cache line for a single coalesced GPU operation, a 64B cache line for an x86_64 CPU, and the maximum transmission unit, which is typically 4KB, for RDMA) we can guarantee if the maximum size is M there are precisely $\lceil n/N \rceil \times \lceil N/M \rceil$ memory transactions.

In the following subsection, we cover how to perform insertions and removes in a fully atomic regime, where sets and gets are a single atomic operation. In the subsection after that, we relax guarantees to elementwise atomicity.

Fully Atomic Regimes

In a fully atomic regime, where gets and sets are atomic across all elements of a c-object, we aim to retain the internal ordering of our nodes. This enables us to partition nodes quickly and simplifies our algorithms. In the following pseudocode, we will assume sequential consistency of our operations for simplicity. However, in practice, we relax consistency when

possible and intend to insert only the most necessary fences.

Algorithm 16 Fully Atomic Lazy Linked List Insertion

```
1:  while true do
2:      result, prev, curr = search(elm)
3:      if result == FOUND then
4:          return false
5:      end if
6:      prev_lock = lock_guard(prev)
7:      prev_copy = get(prev)
8:      if result == GT then
9:          curr_lock = lock_guard(curr)
10:         curr_copy = get(curr)
11:         if validated(curr, curr_copy, prev_copy) then
12:             if curr_key.has_capacity() then
13:                 curr_copy.insert_in_order(elm)
14:             else
15:                 next_copy = curr_copy.partition(elm)
16:                 curr_copy.insert_in_order(elm)
17:                 new_node = new node()
18:                 next_copy.next = curr_copy.next
19:                 set(new_node, next_copy)
20:                 curr_copy.next = new_node
21:             end if
22:             set(curr, curr_copy)
23:             return true
24:         end if
```

In this regime, we first will search in Algorithm 16. Once we reach our node, we will lock the previous and the current node. We will validate that the current node is reachable from the previous node, that the previous node has elements less than our key, and that the

```
25:     else
26:         if validated(curr, curr_copy, prev_copy) then
27:             if prev_copy.has_capacity() then
28:                 prev_copy.insert_in_order(elm)
29:             else
30:                 next_copy = empty_node().insert_in_order(elm)
31:                 new_node = new node()
32:                 prev_copy.next = new_node
33:                 set(new_node, next_copy)
34:             end if
35:             set(prev, prev_copy)
36:             return true
37:         end if
38:     end if
39: end while
```

current node has keys greater than our key. Once this is confirmed, we can modify a copy of our c-object or allocate a second node and partition our node. Once done, we can store it. The benefit of this regime is that we only need two sets at maximum in the critical section. Extending our analysis from before, where we have a list of n elements with N elements per node, this means that in the worst case, we must execute $\lceil n/N \rceil + 2$ operations.

If we can only coalesce M elements maximum, then there are $\lceil n/N \rceil \times \lceil N/M \rceil + 2\lceil N/M \rceil$ memory transactions. In this algorithm, we only need to allocate if the node is full. Since our nodes are N elements, we amortize this allocation across multiple operations.

We remove an element from the list in Algorithm 17. If we traverse and find our key, we similarly validate that the key is present in the node and that both previous and current are reachable. Once we validate, we remove the element and sort the internal node. We then can delete the node if it is now empty. In our removal algorithm, we only need two sets maximum and have the same bounds as the insertion algorithm. This only occurs when a node is empty, meaning we essentially amortize the removal cost.

Elementwise Atomic Regimes

Relaxing the assumptions to elementwise atomicity, the set and get operations are not guaranteed to be atomic across the entire node. We similarly wish to maintain internal sorting of the nodes for faster partitioning of nodes and internal lookup. As we wish to

Algorithm 17 Fully Atomic Lazy Linked List Remove

```
1: while true do
2:     result, prev, curr = search(elm)
3:     if result != FOUND then
4:         return false
5:     end if
6:     prev_lock = lock_guard(prev)
7:     prev_copy = get(prev)
8:     curr_lock = lock_guard(curr)
9:     curr_copy = get(curr)
10:    if validate(prev_copy, curr_copy, curr) then
11:        curr_copy.remove_and_sort(key)
12:        if curr_copy.empty() then
13:            curr_copy.next.mark_deleted()
14:            set(curr, curr_copy)
15:            prev_copy.next = curr_copy.next.unmarked()
16:            set(prev, prev_copy)
17:        else
18:            set(curr_copy, curr_copy)
19:        end if
20:        return true
21:    end if
22: end while
```

maintain the internal sorting of nodes, we opt to create copies of nodes upon update. For example, if we have a node with only one element, and we wish to insert into this node, we can create a copy of the node in memory and update the previous node to point to our new node. This maintains the lock-freedom of reads but adds extra costs of allocation. However, every modification to our nodes can now be coalesced.

First, we will consider how we may modify Algorithm 17 as it is the simplest to change. Before beginning our operation, we can pre-allocate a node to prevent this operation from being in the critical path. In our snippet of the modifications in Algorithm 18, upon finding the element we wish to remove and validating the nodes, we can update our copy of the c-object for the current node. If the node is empty, we can perform a single atomic store on the current node's next pointer to mark the current node deleted, followed by an atomic store to update the previous node's next pointer. If our node is not empty, we can update our new current through a set and atomically store it to set the previous node's next pointer. The critical section of the removal now requires two non-coalesced

Algorithm 18 Elementwise Atomic Lazy Linked List Removal

```
1: / / ...
2: if validated then
3:     curr_copy.remove_and_sort(key)
4:     if curr_copy.empty() then
5:         next = curr_copy.next.mark_deleted()
6:         curr.atomic_store(index_of_next), next)
7:         prev.atomic_store(index_of_next, next.unmarked())
8:     else
9:         set(new_curr, curr_copy)
10:        prev.atomic_store(index_of_next, new_curr)
11:    end if
12:    return true
13: end if
14: / / ...
```

operations, or one coalesced operation and one non-coalesced operation. This means that the number of memory transactions in the worst case using the same variables as above is $\lceil n/N \rceil \times \lceil N/M \rceil + \max(\lceil N/M \rceil + 1, 2)$. While this is less complexity when $M > N$ than in the fully atomic regime, we are less efficient in accessing memory since we cannot coalesce as many operations.

Similarly, for Algorithm 19, if we aim to have internally sorted nodes, we require every update to the current node to be done with a new allocated node as well as an extra modification of the previous node's next pointer. This means we require three writes in the worst case and two at least upon a successful insert. At least one of these writes cannot be coalesced. The worst-case complexity of this operation is $(\lceil n/N \rceil + 2) \times \lceil N/M \rceil + 1$. This means our updates are always more costly than in the fully atomic regime, but only by one extra memory transaction.

In practice, we find that this regime has some overhead. However, by approaching the problem through this kind of breakdown of memory operations and transactions, a well-thought-out algorithmic design can minimize these costs.

We implement generic versions of the lazy linked list with CODS in C++, which can run on a CPU, GPU, or over RDMA. We enable toggling full atomicity, changing the number of threads performing coalescence on the GPU, changing vectorization size, and varying the node size. We then utilize this lazy linked list to implement a hash set.

Algorithm 19 Elementwise Atomic Linked List Insertion

```
1: // ...
2: if return == GT and validated then
3:     if curr_key.has_capacity() then
4:         curr_copy.insert_in_order(elm)
5:     else
6:         next_copy = curr_copy.partition(elm)
7:         curr_copy.insert_in_order(elm)
8:         new_node = new node()
9:         next_copy.next = curr_copy.next
10:        set(new_node, next_copy)
11:        curr_copy.next = new_node
12:    end if
13:    set(new_curr, curr_copy)
14:    prev.atomic_store(index_of_next, new_curr)
15:    return true
16: end if
17: // ...
18: if return == END and validated then
19:    if prev_copy.has_capacity() then
20:        prev_copy.insert_in_order(elm)
21:    else
22:        next_copy = empty_node()
23:        next_copy.insert_in_order(elm)
24:        new_node = new node()
25:        prev_copy.next = new_node
26:        set(new_node, next_copy)
27:    end if
28:    set(new_prev, prev_copy)
29:    prev_prev.atomic_store(index_of_next, new_prev)
30:    return true
31: end if
```

With CODS, we also implement a B-link tree. The B-link tree differs from a traditional B-tree in how it splits a node through two steps. In the B-link tree, splitting is done first as a simple atomic operation followed by inserting a link to the child in the parent node rather than locking the entire structure. A link between the split nodes allows for consistent traversal during the split. Our implementation only supports gets and inserts. Our approach is inspired by Awad et al.'s GPU B-Tree [21], which implements a B-link tree on the GPU and relies on replacing nodes in the B-link tree with c-objects similar to what was done for linked lists. The following section evaluates our CODS implementations of these CPU, GPU, and RDMA data structures.

5.5 Evaluation

We evaluate CODS for CPU, GPU, and RDMA by implementing and measuring performance on node-based data structures. Our evaluation aims to demonstrate the applicability of CODS to all three architectures, demonstrate that it is possible to implement existing work in literature with CODS, and understand the practical performance implications of node size and fully atomic versus elementwise atomic regimes. We specifically focus on the linked list designed above, chaining hash sets, and B-link trees [21].

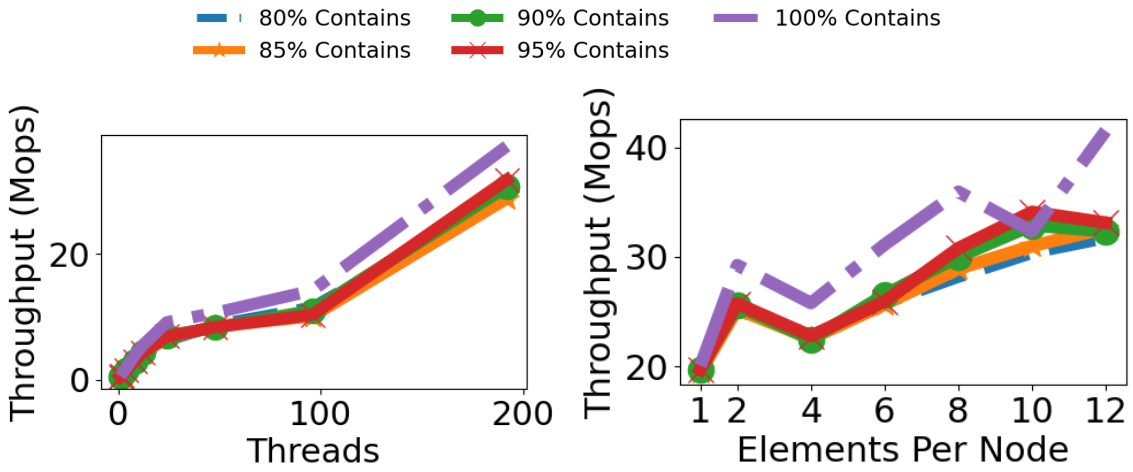
We implement CODS in C++ and build with g++-11 and the 12.6 CUDA toolkit for GPU support. For our RDMA support, we utilize an existing experimental library called Remus [24], which wraps RDMA verbs calls. For our CPU evaluation, we utilize a test bed with 4 Intel Xeon Platinum 8160 CPUs with hyper-threading enabled, giving us 96 cores with 192 hardware threads. For the GPU and RDMA-based evaluation, we benchmark on Cloudlab [59]. The GPU evaluation is done on a Cloudlab r7525 node with one of the 32 GB NVIDIA Tesla V100S PCIe GPUs. The RDMA evaluation is done on a Cloudlab r320 node with an 8-core Xeon E5-2450 and Mellanox Connect X-3 card with 10 Gb/s RDMA over converged Ethernet (RoCE). All evaluation is done within a closed loop. Throughout the evaluation, we select a key range and pre-populate the data structure to half of the key range.

We divide our evaluation into three subsections: Section 5.5.1 for CODS on the CPU, Section 5.5.2 for CODS on the GPU, and Section 5.5.3 for CODS on RDMA clusters.

5.5.1 CPU

We begin by evaluating the application of CODS on the CPU. In practice, we found minimal difference between the fully and elementwise atomic regimes, so we omit this in assessing the CPU CODS-based data structures. We utilize the linked lists algorithms described in Section 5.4. For our c-objects, we use a vector size of 1 element but allow the compiler to optimize memory operations into SIMD instructions. Regardless of whether we set the vector size to 1 or higher, the compiler will vectorize loads and stores with SIMD instructions. We evaluate the CPU linked list with a range of 2000 and prepopulate to 1000

elements before running.



(a) Scalability of 8 element nodes.

(b) Node Size at 192 Threads.

Figure 5.1: CODS CPU Lazy List

In Figure 5.1a, we look at the scalability of the linked list implementation with 8 element nodes. We run experiments from 80% contains operations to 100% contains, where the rest of the operations are evenly divided between insertions and removal. We pin threads to the first CPU for threads 1 to 24, then pin to the next core for the next 24 threads; once we reach 96 threads, we enable hyperthreading.

We find a 21% and 46% gap in throughput between 80% reads and 100% reads, with performance generally increasing as the percent of reads increases. The gap is due to the modifications, which must lock nodes and block execution.

We further evaluate by varying the node size in Figure 5.1b. We find in practice that a node size of 8 is a practical-linked list size, at 10 elements for workloads in the 90% to 95% contains range plateau. At the same time, performance continues to increase for more update-heavy workloads up to the performance of the 90-95% update workloads. Read-only workloads are subject to the performance impacts of vectorization. While there is a generic trend upward for 100%, we believe the dips are related to dynamic voltage-frequency scaling impacts from executing SIMD instructions, which are more evident since the CPU is not blocked on locks.

There is no clear conclusion for an optimal node size for all workloads for this workload; instead, this suggests that node size should be tuned. However, when well tuned, there is

a 42% to 79% performance gain from utilizing coalescence. Since CODS is agnostic to the node size, CODS can enable the design and tuning of data structures for each workload.

Next, we evaluate CODS on a chaining hash set. We set the size of the hash map to 125,000 and use a range of 1,000,000 elements. We prepopulate to 500,000 elements.

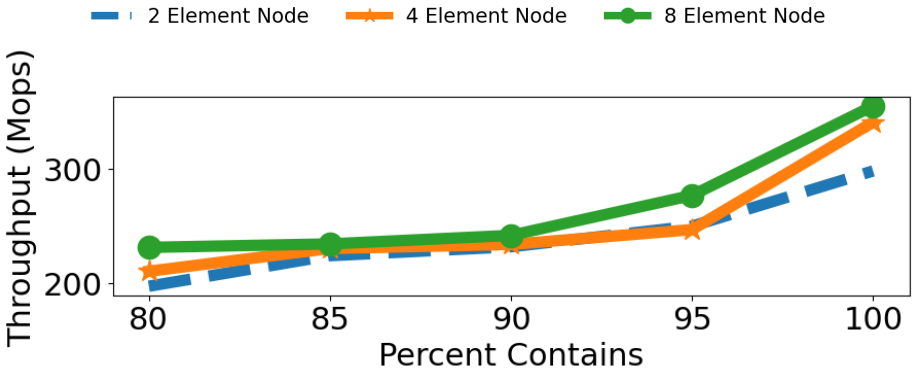


Figure 5.2: Plot of CPU Hash Set Varying Percent Contains.

We find that performance generally increases for the hash set implementation as the workload becomes more read-heavy. Furthermore, increasing the elements per node is a worthwhile optimization as it enables the ability to scale the number of elements placed in one bucket. An expected load factor of 4 means that the nodes are expected to contain 4 elements, and with a larger node size, a single read is all that is necessary to read the data in the bucket. This effectively improves performance from 4% to 19% when moving from 2-element nodes to 8-element nodes.

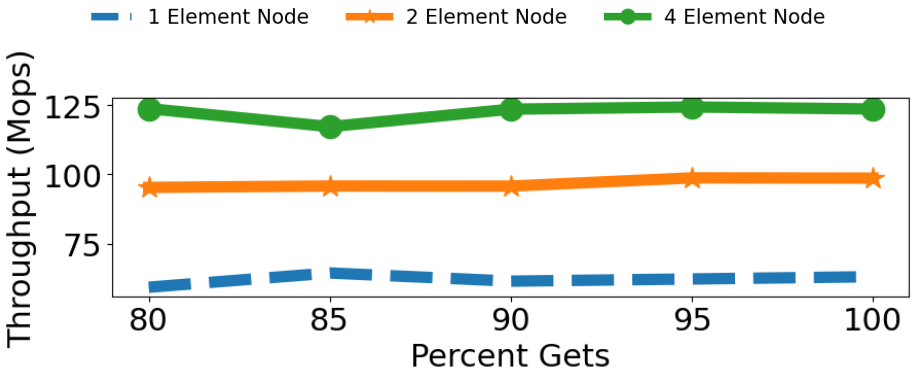


Figure 5.3: CPU B-Link Tree Throughput While Varying Percentage of Gets

Finally, we evaluate our B-link tree implementation with CODS on the CPU. We use

a range of 1,000,000 keys and prepopulate to 500,000 keys. We find that increasing the number of insertions affects our B-link tree implementation less than the linked list or hash set. It is also worth noting that the B-link tree does not perform as well as the hash map due to the ordering constraints. Similar to the hash map, increasing node size can positively impact performance. When switching from 1-element nodes (a balanced tree) to 2-element nodes, we find a 49% to 61% speedup. From 2-element nodes to 4-element nodes, we observe a 22% to 30% speedup.

5.5.2 GPU

We evaluate CODS on GPUs by looking at the linked list implementation, a hash map, our B-link tree, and porting one of the B-link trees from [21]. We evaluate the CODS data structures in a batch synchronous manner, with a stream for each batch of operations. This enables the driver to schedule the kernels concurrently if available resources exist. Unless mentioned otherwise, we execute 4,000,000 operations where some percentage is read-only, and the other fraction of the operations modifies the data structure.

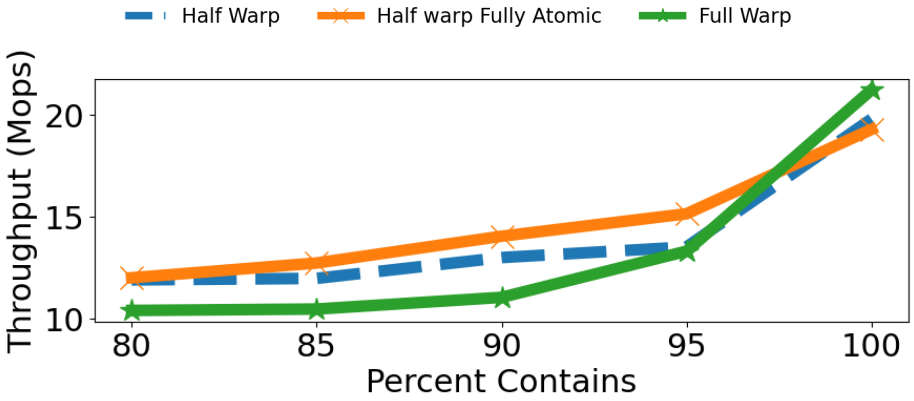
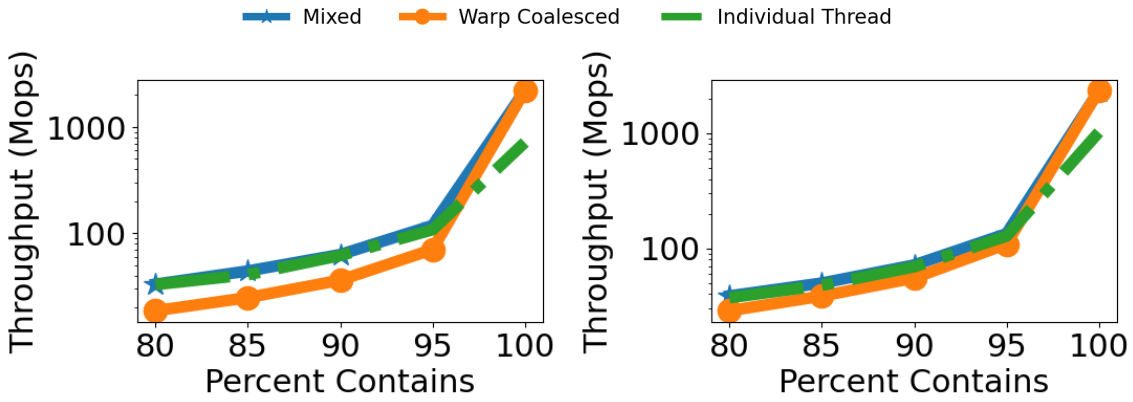


Figure 5.4: CODS GPU Lazy Linked List

We begin with evaluating our GPU linked list in Figure 5.4 with a key range of 20,000 and prepopulated to 10,000. We compare a full warp, a half warp, and a half warp with fully atomic execution. We find both half warp versions outperform the full warp version for the 80% to 95% contains workloads. This is due to the extra parallelism that half-warps provide. When there is contention for a warp, 32 threads are blocked, but in the half warp regime, the other 16 threads may be able to do some work. We find that a fully atomic half

warp has a 14% to 27% speedup over using a warp, and a half warp generally has a 2% to 18% speedup over a warp. At 100% reads, the full warp can better coalesce accesses and can achieve a 10% speedup over the fully atomic half warp implementation.



(a) Node size = 30

(b) Node size = 14

Figure 5.5: GPU Hash Set

In Figure 5.5, we evaluate a GPU implementation of a hash set. The hash set design is inspired by SlabHash [18] and L-Slab [53], where a warp cooperative approach is used to reach optimal performance. Compared to the prior approaches, we use our linked list, which has a bit set to denote whether key-value pairs are present in the data structure rather than a specific key (e.g., 0) to denote that a key-value pair is not present in a node.

Due to the generality of the CODS programming model, we test both 30-element nodes with a bit set and next pointer and 14-element nodes with a bit set and next pointer, as well as using single-threaded or warp-based execution. In the single-threaded mode, we rely on the compiler to vectorize accesses or from the hardware to benefit from spatial locality. We set the range to 1,000,000 elements and prepopulate to 500,000 elements. For the 30-element nodes, we use 33,333 buckets, and for the 14-element nodes, we use 71,428 buckets.

From this evaluation, we discover that warp-coalesced accesses are not always optimal. In fact, through warp coalescence, we induce many points where threads must synchronize with one another within a warp through register shuffles, which increases stalling. With only one load per element, the compiler cannot effectively pipeline loads. In the single-threaded execution, however, the compiler is able to effectively pipeline loads.

Comparatively, for modifications, warp and half-warp coalesced approaches reduce parallelism by a factor of 1/32 or 1/16, respectively. When a group of threads becomes blocked waiting on a lock, progress cannot be made for any of the threads; however, if a single thread becomes blocked, other non-contending threads may progress, increasing throughput.

To better equip ourselves to alleviate this bottleneck, we introduce a mixed approach utilizing CODS. We execute with a single thread for our kernels that perform modifications, but for gets, we execute in a warp-coalesced fashion. The mixed approach maintains optimal performance.

The mixed approach outperforms the warp coalesced approach by 68% to 78% in Figure 5.5a workloads with updates. When changing the node size to 14 elements per node with a fully atomic regime, we find a speedup of 23% to 35%. Compared to the single-threaded approach, the mixed approach can maintain higher performance in read-only workloads because its get operations utilize warp coalescence with 3.1x and 2.3x for 30-element and 14-element nodes, respectively.

When moving from 30-element nodes to 14-element nodes with full atomicity, we find an average performance improvement of 12%. Furthermore, we note that, this is similar to what we observe with the linked list.

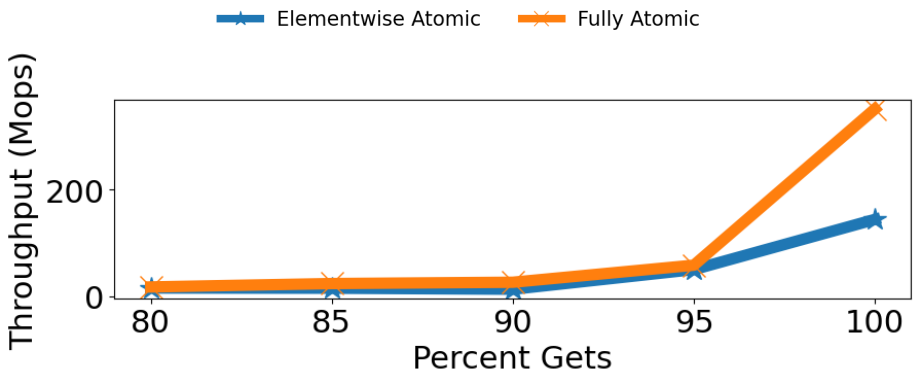
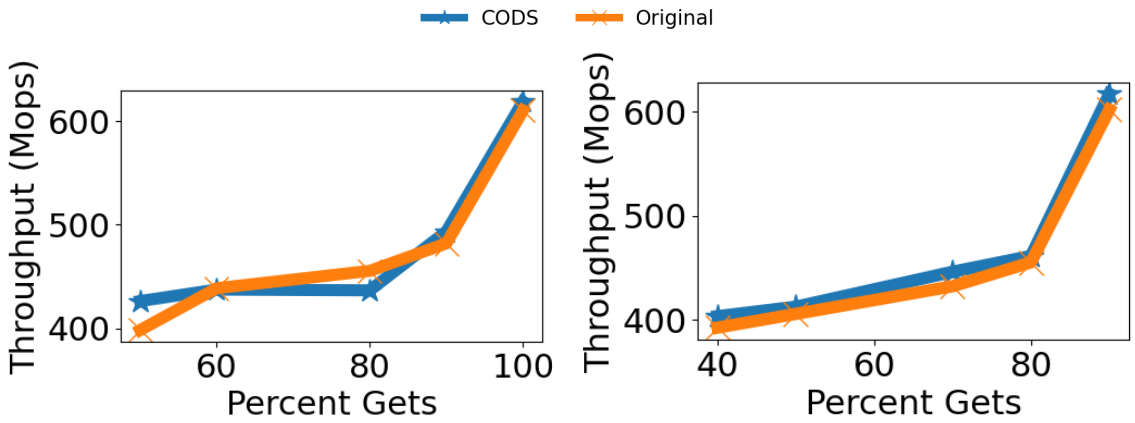


Figure 5.6: CODS GPU B-Link Tree

We implement a fully atomic and elementwise atomic B-link tree in CODS for the GPU. In Figure 5.6, we plot warp-coalesced versus non-warp-coalesced approaches. We use a key range of 1,000,000 and prepopulate to 500,000. From this experiment, we can conclude that coalescence also helps GPU B-trees. The speedup ranges from 13% to 2.4x. Full atomicity

is especially impactful in read-only scenarios for the B-link tree, unlike the linked list, where there is a 2.4x speedup.



(a) 0% range queries

(b) 10% range queries

Figure 5.7: Awad et al.'s B-Link Tree With And Without CODS; Varying Gets

We also add CODS to an existing GPU data structure to demonstrate that it can achieve performance similar to existing work. In Figure 5.7, we show CODS applied to the multi-versioned GPU data structure from [21]. In this paper, Awad et al. utilize coalescence to optimize the GPU performance of B Link Tree. Compared to implementing a B-link tree Awad et al. make a few specific optimizations. This includes assumptions about the key range so they can utilize the max key to mark it as deleted. However, our approach utilized a bit set that is broadcasted throughout the warp to mark keys as present. Awad et al. also utilize a back-off mechanism for executing within their B-Tree to minimize lock contention, which we do not implement. This data structure utilizes a special lock implemented through the atomic or operation implemented in CUDA. We do not use this operation, but it enables Awad et al. to coalesce accesses to the lock.

Applying CODS to the B-tree requires 138 lines of code change. This is 9% of the lines of code that implements the B-tree, and most changes are minimal. In practice, the compiled PTX (targetable GPU assembly) and SASS (GPU machine code) differ, but the performance suggests that CODS maintains the original optimizations. We evaluate the B-tree with a key range of 1,000,000 and prepopulate to 500,000. We run both 10% range queries and no range queries, varying percentage gets from 40% to 90% of operations and

50% to 100% of operations, respectively. We evenly divide the rest of the operations between insertions and removals. In many cases, in Figures 5.7a and 5.7b, CODS enables a 1% to 3% speedup over the original implementation and is consistently within 4% of performance of the original implementation.

5.5.3 RDMA

Finally, we evaluate CODS using an RDMA-based system. We have a 10 Gb/s network over Ethernet connected with only a single switch between the nodes. In this evaluation, we are limited by how much memory the cluster would let us register as RDMA accessible. We can register 256 MB per thread per server if we utilize 4 threads per server (1 GB per node). We must utilize distributed allocation mechanisms to effectively manage memory in an RDMA system, which could impact performance. To focus on evaluating the coalescing optimization's performance, we leak memory related to the data structure when it would require extra coordination. Due to these constraints, we focus on smaller data structure sizes than evaluated for the CPU and GPU. To ensure consistent performance, we divide the population across all nodes to ensure the workload is balanced when we prepopulate. All experiments for CODS on RDMA are conducted with 4 threads per node executing requests.

We first evaluate our linked list in Figure 5.8. We use a range of 2,000 keys and a population of 1,000. We find that the results are fairly similar for any workload with any concurrent modifications, so we focus on 95% reads. Within our linked list, 1 to 6 elements per node have consistent performance. These node sizes allow for full atomicity because they can fit into a 64B cache line with a bitset and next pointer. With these smaller nodes, increasing the number of servers leads to increased performance. From 2 to 5 servers, we observe a 72% to 81% increase in throughput. Moving from full atomicity with 6 elements per node to elementwise atomicity with 8 elements per node results in a significant performance degradation that is more impactful the more servers present in the system. There is a 62% decrease in performance from 2 to 5 servers. For this workload, full atomicity while scaling out servers is ideal.

We evaluate a hash set in Figure 5.9 with a key range of 100,000 keys. For this evaluation,

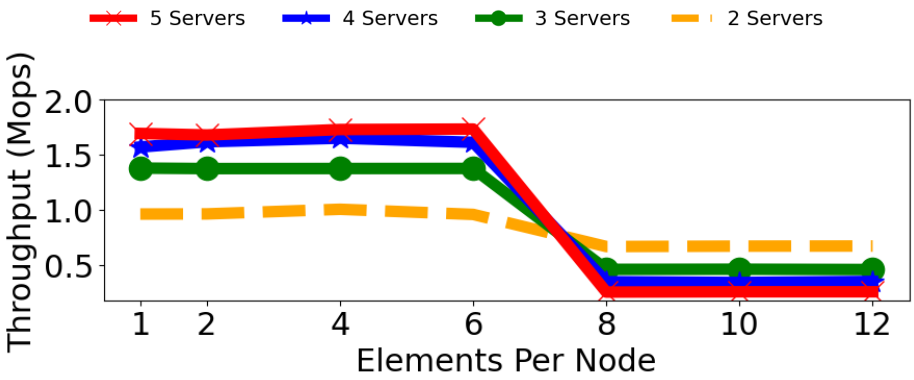


Figure 5.8: RDMA Lazy Linked List; 95% Contains, 2.5% Inserts, 2.5% Removes.

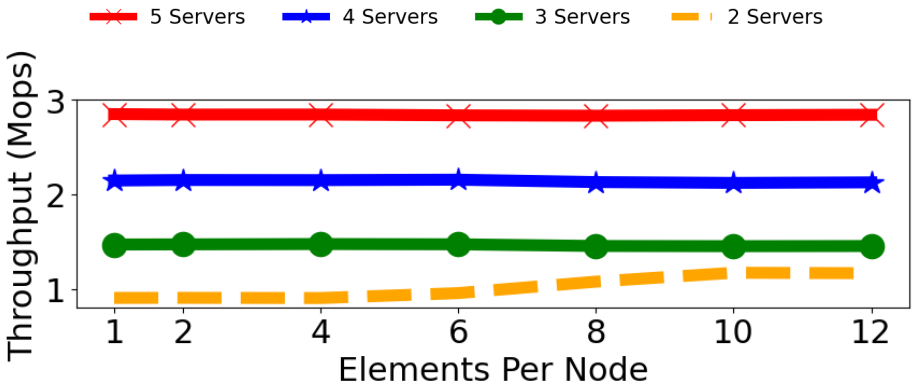


Figure 5.9: RDMA Hash Set; 95% Contains, 2.5% Inserts, 2.5% Removes.

we set the size of the hash set to 100,000 as well, for a load factor of 1. Since the load factor is 1, coalescence will only help when chaining is necessary since it will limit the number of traversals needed. Increasing the number of elements per node for two servers impacts performance by increasing the achievable throughput. However, scaling out is a more impactful strategy for a hash set similar to what was observed for the linked list that implements it. Unlike the linked list, full atomicity is unnecessary to achieve optimal performance. Once we scale out, throughput scales almost linearly by 2.4x when going from 2 to 5 servers for 12 elements per node.

We test our B-link tree in Figure 5.9 with a range of 10,000 keys and a population of 5,000. Unlike the hash set, performance increases significantly with the number of elements per node. We attribute this to the fact that interior nodes are typically less likely to be modified, and therefore, the contention is primarily on leaf nodes. For 5 servers, performance increases 2.7x when moving from 2 elements per node to 12 elements per node. When moving

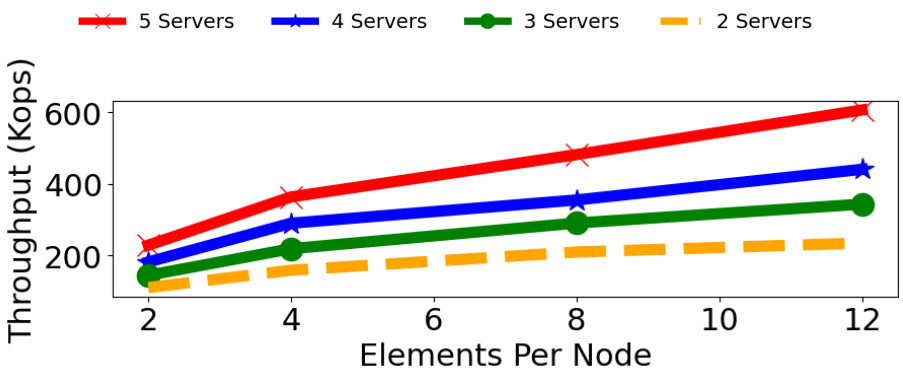


Figure 5.10: RDMA B-Link Tree; 95% Gets, 5% Inserts.

from 2 servers to 5 servers, however, at 12 elements per node, performance increases 2.6x. CODS provides a method to achieve this performance improvement in the B-link tree.

5.6 Summary

This chapter presents CODS, a framework that generalizes the coalescence optimization for data structures across CPUs, GPUs, and RDMA. Coalescence is a powerful optimization applied in a variety of data structures, including CPU data structures [156, 157, 147, 163], GPU data structures [21, 23, 18, 53, 22], and RDMA data structures [181, 164, 182, 58, 131]. We implemented prior works using the CODS framework, including a lazy linked list based on Heller et al.'s design [87], a B-link tree [21], and modifies Awad et al.'s B-Tree [21] to use CODS.

CODS's work is orthogonal to ongoing work in multi-versioning data structures [184, 29, 130]. This work is utilized by Awad et al. in their multi-versioned B-link tree [23] and by Nelson-Slivon et al. in their RDMA B-Tree [131].

We demonstrate that CODS can implement high-performance data structures and enables programmers to fine-tune coalescence to their data structure workload based on trade-offs between fully and elementwise atomicity, number of elements per node, and warp or individual thread-based coalescence.

Chapter 6

Programmable Range Queries on CPU and GPU Through the Lookup Interlocked Table

6.1 Introduction

Range queries on *map* data structures are operations that iterate over the map and return a collection of key-value pairs where the key falls within a contiguous range. They are emerging as an essential API for data repositories with key-value semantics, including Redis [155], RocksDB [57], LevelDB [76], and others [61, 197, 146, 99], and for optimizing the internals of relational databases, which traditionally perform range queries through predicates [167]. Their popularity has also extended beyond traditional CPU-based systems. In fact, data structures supporting range queries have been deployed on heterogeneous processors, such as the GPU [6, 23, 21], and integrated into real-world products, such as OmniDB [194] and RateupDB [107]. In this chapter, we are interested in designing a new data structure that provides range queries and whose principles show affinity to both CPU and GPU.

Highly concurrent data structures that support range queries typically implement operations through costly traversals over linked data, such as the case of linked lists [40, 184, 16, 130], skip lists [40, 16, 130, 7], or B-trees [166, 21, 23]. However, it is well known that

if range queries are not necessary, hash tables [88] provide a method to limit the overhead of traversals by partitioning elements into *buckets* and index into them by means of a hash function that redirects the operation to the bucket where the requested element is stored. Because of the absence of long traversals and minimal housekeeping overhead, hash tables have been shown to scale better than linked data structures and provide higher performance under a variety of workloads [77].

It is natural, then, to wonder if retaining the hash table structure's performance advantage is possible while still providing ordering between elements, which is important for implementing consistent range query operations. Motivated by the above observation, we present the *lookup interlocked table* (or LIT) in this chapter. LIT has the structure of a traditional hash table [88], where each bucket is implemented as a linked list of elements. However, rather than having a hash function that uniformly hashes keys to buckets, we utilize a *lookup* function that maps keys to buckets *and* preserves the ordering of keys (i.e., if $x < y$, then $f(x) \leq f(y)$, where x and y are keys).

In LIT, every operation starts by invoking a lookup function that identifies the bucket where the relevant element is located. This workflow applies to elemental operations (i.e., add/remove/contains) and range queries, where the lookup function determines the starting point of the range. Subsequent buckets are visited for range queries by utilizing an overlay linked structure that connects all elements of all buckets.

To efficiently handle changing workloads, LIT takes inspiration from the interlocking structure of the IHT table [96]. LIT operates through buckets that can map to another level of buckets, which helps reduce the number of traversals necessary to reach a node in case many elements are inserted into a single bucket. This operation is called a `SPLIT`. Over time, many split operations to a single bucket might lead to an increase in the traversal cost. LIT addresses this issue by efficiently flattening the structure to a single level to restore the original indexing performance and re-balancing elements. We call this operation `RESIZE`. Resizing is non-blocking and is performed efficiently by taking advantage of the overlay-linked structure and the invariant enforcing ordering between buckets.

Our design implements the overlay linked structure using vCAS [184], a state-of-the-art technique that provides non-blocking range query operations over existing lock-free data

structures. Essentially, vCAS adds versioning to the linked lists, allowing traversals to observe a consistent view of the structure.

The LIT design focuses on minimizing pointer chasing, which is beneficial not only for CPU execution but also for heterogeneous devices that obey to the lock-step execution model, in which all threads execute the same instructions on multiple input data. This is the case of GPGPUs (General-Purpose Graphics Processing Units). GPUs are readily available and provide cost-efficient hardware parallelism, making them appealing for improving the performance of highly concurrent data structures. An increasing number of products [28, 86, 177, 194, 107, 173, 141] rely on GPU-based data structures to speed up searches and analytics.

Adapting LIT to GPUs is done by extending the data structure to include multi-element nodes, similar to those present in B-trees [23, 21]. This enables us to coalesce memory accesses and utilize well-known GPU-specific synchronization and cooperation techniques [18, 23, 21, 53] to orchestrate the execution of GPU threads.

We implement LIT in C++ and CUDA and compare it against state-of-the-art data structures supporting range query operations through Bundling [130] and vCAS [184]. LIT can achieve up to a 2.8x speedup over versioned skip list approaches and 2.2x over a versioned CITRUS tree [15] in common zipfian workloads using 80% get, 10% update, and 10% range query operations. On the other hand, although fast, GPU-based state-of-the-art competitors are unable to index data structures with key-value pairs larger than 4 bytes. Our ported design of LIT allows for that while retaining high performance.

The rest of the chapter is organized as follows. In Section 6.2, we detail the layout of the LIT data structure and how it supports point operations and range queries. In Section 6.3, discuss the split and resize operations. Section 6.4 describes how we extend the design of LIT to GPU. Then, in Section 6.5, we evaluate LIT against state-of-the-art competitors and finally Section 6.6 overviews the related work and Section 6.7 concludes the chapter.

6.2 LIT: Lookup Interlocked Table

The lookup interlocked table (LIT) is a key-value mapping data structure with the following operations:

- `GET`(key): returns the associated value or none if the key is not in the map.
- `INSERT`(key, value): if the key is not in the map, it inserts the key-value pair and returns true; if the key is in the map, it returns false.
- `REMOVE`(key): removes the key-value pair from the map if it is present.
- `RANGE_QUERY`(start, end): returns key-value pairs within the provided range of keys.

All operations in LIT are linearizable. This means that they appear to occur atomically between the invocation and return. LIT uses a structure similar to the chained bucket structure of a hash table [118], along with a function that preserves the order of the keys. When initialized, LIT is given a size corresponding to the number of buckets, a predicted minimum, and a predicted maximum element.

In addition to partitioning elements into buckets, LIT allows each bucket to be further split into more buckets recursively (interlocking design [96], more details later). Every time a `SPLIT` occurs, a new level in LIT is created. Similar to the terminology used in skip lists [51], we call the collection of those levels the *index layer*. However, unlike a skip list, LIT utilizes a collection of lookup functions to map an element into the index layer, one per level. Each lookup function f preserves the order of keys in its level such that if $x < y$, then $f(x) \leq f(y)$. In principle, any function that preserves this property works.

For a finite key space, we utilize the following function that interpolates a set $[a, b)$ over N buckets of the form:

$$f(x) = \text{clamp}(\lfloor \frac{N}{d(a,b)} d(a,x) \rfloor, 0, N-1)$$

where:

- `clamp` is a function that returns 0 if the first argument is less than 0, $N-1$ if the first argument is greater than $N-1$, and otherwise returns the first argument;

- b is the predicted max of the given level's range;
- a is the predicted min of the given level's range;
- d is a distance function for which $d(x, y) = |\{h \in U : y > h \geq x\}|$ where U is the universe of keys (i.e., the distance between two keys is equal to the cardinality of the keys in that range). For integers, we utilize $d(x, y) = |x - y|$ as our chosen distance function. For floating point numbers, we calculate the number of floating point numbers between two numbers. This can be done simply by reinterpreting two floating points of the same sign as integers and subtracting them. For other universes in which this distance function is unknown, we suggest choosing a lookup function so that d provides a measure of the density of keys in a set (e.g., length or volume of the set).

It is clear that f preserves our ordering property since it increases from 0 to $N - 1$ as x increases ($a \leq b$ and N is a positive non-zero integer). As an example, if we assume LIT has four buckets, a single level, and a predicted range of integers 0 to 7 (similar to the one in Figure 6.2a), then

- elements with keys ≤ 1 maps to Bucket 0;
- elements with keys ≥ 2 and ≤ 3 maps to Bucket 1;
- elements with keys ≥ 4 and ≤ 5 maps to Bucket 2;
- elements ≥ 6 maps to Bucket 3.

Figure 6.1 depicts this function.

Since elements within a bucket are ordered and elements in subsequent buckets are also ordered due to the just described lookup function, we can interconnect all these nodes, forming a linked list that we call *data layer*. To point a bucket to a location in this data layer, LIT uses special nodes called *boundary-markers*. These nodes have a key that matches the minimum key of the bucket they are linked to. For example, if we consider a LIT with a single level that contains elements with keys "0", "2", "3", "4", "7", and "9," (assuming the lookup function from Figure 6.1) then we have the structure depicted in Figure 6.2a.

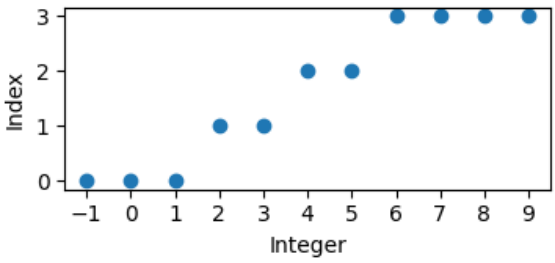
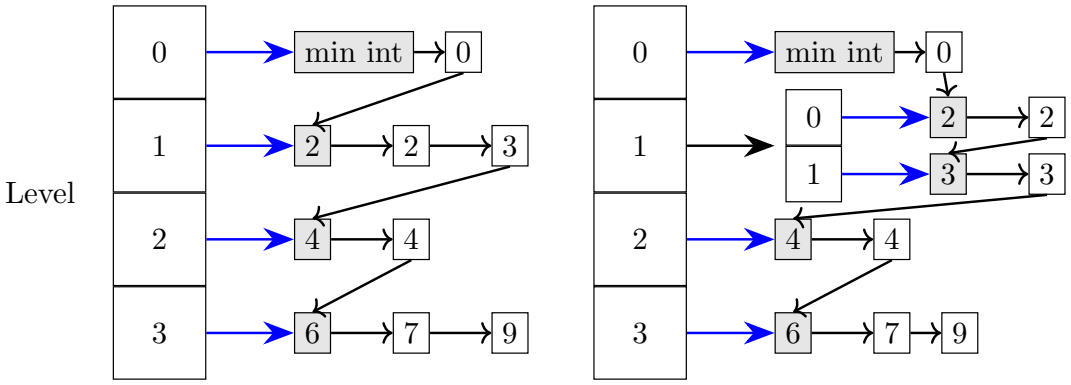


Figure 6.1: A lookup function ($a = 0$, $b = 7$, $N = 4$) which maps integers to indices in a level.



(a) Example the LIT Data Structure; boundary-markers are colored in gray; Blue arrows are from the index to the data layer.

(b) Example of LIT with two levels; Boundary-markers are colored in gray; Blue arrows are from the index to the data layer.

Figure 6.2: LIT's structure for elements with keys "0", "2", "3", "4", "7", and "9," before and after splitting Bucket 1.

Here, the boundary-markers are: "min-int" (i.e., the minimum representable integer), "2", "4", and "6". Each bucket points to its own boundary-marker. If an element with that key is then inserted into the data structure, a new node with that key will be placed right after the boundary-marker. In Figure 6.2a, this is the case of Bucket 1, in which key "2" is placed right after the boundary-marker with key "2". The nature of this data layer allows for traversals when concurrent modifications occur (more details below).

In traditional hash tables, a good hashing function guarantees that elements are balanced among buckets. Since LIT does not rely on hashing, we deploy a different technique, which enables splitting buckets into other levels. In Figure 6.2b, a split operation divided Bucket 1 into two buckets. Note that upon the bucket split, similar to Figure 6.2a, a boundary-marker node is created for each bucket in the new level. Now, if an operation is looking

for key "3", that operation should invoke the lookup function twice, one for Level 0, with parameters $b = 7$, $a = 0$, and $N = 4$, and one for Level 1 with parameters $b = 3$, $a = 2$, and $N = 2$. We detail how splits occur and its correctness and liveness guarantees in Section 6.3.

In the remainder of this section, we detail how LIT implements its operations. For clarity, throughout the next subsections, we assume *no concurrent resizes or splits occur*. Then, in Section 6.3, we address how LIT concurrently splits buckets and resizes the entire structure.

6.2.1 Bucket Lookup

All LIT operations, both elemental operations (i.e., `INSERT`, `REMOVE`, and `GET`) and range queries, start by invoking a search function that uses the index layer to reach the proper entry point to the data layer by returning a pointer to the boundary-marker of the bucket to start with. For elemental operations, this is the bucket to which the element belongs, and for range queries, this is the bucket to which the element with the smallest key belongs. Algorithm 20 overviews the procedure to achieve that. Starting from Level 0 (i.e., the initial set of buckets). First, we read the root level and then utilize the level's lookup function to find the subsequent level. We repeat this until we find a pointer to a boundary-marker ordered before the key we seek. LIT guarantees this is always a traversal less than a threshold T_l, at which point it will resize (more in Section 6.3).

Algorithm 20 LIT Bucket Lookup

```
1: Given: Key k
2: level = root
3: idx = level→f(k)                    ▷ f is our lookup function for level
4: while level[idx] is not a linked list do
5:     level = level[idx]
6:     idx = level→f(k)
7: end while
8: return level[idx]
```

As noted earlier, this search algorithm assumes that no simultaneous modifications are being made to the index. Without split and resize operations, boundary-markers cannot be removed while traversals are executed, and the number of levels for each bucket cannot change. In Section 6.3, we introduce a modified version that can handle concurrent `SPLIT`

and `RESIZE` operations.

6.2.2 Bucket Operations

Starting from the boundary-marker returned from the above search method, multiple techniques can be potentially used to perform the operations on the data layer.

Recent literature has proposed effective techniques to enable linearizable range queries for concurrent data structures while other operations can concurrently modify disjoint data [184, 16, 130]. We choose to use vCAS [184] because, unlike other blocking techniques [16, 130], it introduces wait-free range queries to lock-free data structures. In our implementation, we port vCAS to the Harris lock-free ordered linked list [85], as originally done by vCAS [184] as well, with the necessary modifications to consider the fact that our list traverses through buckets of an enclosing index. The correctness of our approach in the absence of `SPLIT` and `RESIZE` relies on the correctness of vCAS and the Harris list.

vCAS in a nutshell

In a vCAS data structure, all pointers updated using atomic operations are changed to be versioned CAS objects. These version CAS objects contain a pointer to a versioned node (VNode) and a reference to a global timestamp called a Camera. Each VNode contains a value (e.g., the next linked list node pointer), a timestamp, and pointers to earlier VNode versions of the same object. When modifying a data structure to use vCAS, all CAS operations are replaced with performing a CAS on the root VNode of the vCAS object. All reads are replaced with reading the latest VNode head of the vCAS object. Our implementation of vCAS in LIT changes the indexing nodes in the index layer and the Harris list's nodes in the data layer to be vCAS objects.

vCAS is designed to implement logical *snapshotting* of a data structure by incrementing a global timestamp through a Camera object. Incrementing the timestamp and timestamping each VNode enables vCAS to implement multi-versioning of the underlying data structure. In LIT, we modify vCAS to increment the global timestamp on each CAS operation, similar to Bundling [130]. This enables logical snapshots of the data structure to read the global timestamp rather than incrementing it to limit the negative side effects of atomic operations,

especially in non-uniform memory access (NUMA) machines.

Insert/Remove/Get Operations

Algorithm 21 vCAS Lock Free Linked List Search

```
1: Given: Key k
2: prev = nullptr, curr = root.load(), next = nullptr                ▷ Use vCAS load
3: if curr.ptr() = nullptr then
4:     return {prev, curr, next, HEAD_IS_NULL}    ▷ Head is null is an enum to denote status
5: end if
6: while true do
7:     next = curr→next.load()
8:     if curr.markings() & DELETED then    ▷ Check stolen bits since we mark nodes this way
9:         cas(prev→next.ptr(), curr, next)
10:        prev = nullptr, curr = root, next = nullptr
11:        if curr.ptr() = nullptr then
12:            return {prev, curr, next, HEAD_IS_NULL}
13:        end if
14:        continue
15:    end if
16:    is_not_boundary = curr.markings() & BOUNDARY = 0 ▷ Check boundary markers
17:    if is_not_boundary and curr.ptr()→key = k then
18:        return {prev, curr, next, FOUND_KEY}                ▷ Enum for finding key
19:    else if curr.ptr()→key > k then
20:        return {prev, curr, next, GT_KEY}                ▷ Enum for greater than key
21:    else if next.ptr() = nullptr then
22:        return {prev, curr, next, AT_END}                ▷ Enum for end of list
23:    end if
24:    prev = curr
25:    curr = next
26: end while
```

Now we summarize the main changes we made to the elemental operations in our vCAS bucket list.

The main addition to our list is the support of boundary-marker nodes. We do that by stealing two bits in the next pointer of each node. One bit is used to mark the node as a boundary-marker, and the other is used to mark a node logically deleted (similar to the Harris list, which uses logical deletion before unlinking the node).

We also support a new operation, `INSERT_BOUNDARY_MARKER` as seen in Algorithms 23 and 24, which will be invoked when splitting/resizing the data structure. This operation

Algorithm 22 vCAS Lock Free Linked Insert

```
1: Given: Key k, Value v
2: while true do
3:     prev, curr, next, result = search(k)
4:     if result = HEAD_IS_NULL then
5:         node = new node(k, v, nullptr)
6:         if vcas(root.ptr(), nullptr, node) then          ▷ Use vCAS CAS operation
7:             return true
8:         end if
9:     else if result = FOUND_KEY then
10:        return false
11:    else if result = GT_KEY then
12:        tocas = prev == nullptr ? root : prev→next
13:        node = new node(k, v, curr.ptr()) — tocas.markings()      ▷ Retain markings
14:        if vcas(tocas.ptr(), curr, node) then
15:            return true
16:        end if
17:    else if result = AT_END then
18:        node = new node(k, v, nullptr) — curr→next.markings()     ▷ Retain markings
19:        if vcas(curr→next.ptr(), next, node then
20:            return true
21:        end if
22:    end if
23:    continue
24: end while
```

performs the standard insert algorithm, but the node is inserted with the boundary-marker bit set.

Figure 6.3 shows an example of one bucket's list in LIT's data layer, whose minimum key is "0". At time t_0, the list is created by inserting a boundary-marker with key "0". Then at t_1, an element whose key is "1" is inserted. At time t_2, another element with key "0" is inserted. At time t_3, key "1" is marked deleted and then removed at t_4 by updating the next pointer of the node with key "0."

Wait-Free Range Queries

Range query operations in LIT follow the pseudocode in Algorithm 27 and utilize the Search function in Algorithm 20. It is worth noting that a range query might need to traverse multiple buckets to collect the queried range. This operation is done by simply continuing to traverse the data layer and crossing the boundary-marker of the next bucket.

Algorithm 23 vCAS Lock Free Linked List Search for Boundary-Marker

```
1: Given: Key k
2: prev = nullptr, curr = root.load(), next = nullptr
3: if curr.ptr() = nullptr then
4:     return {prev, curr, next, HEAD_IS_NULL}
5: end if
6: while true do
7:     next = curr→next.load()
8:     if curr.markings() & DELETED then
9:         cas(prev→next.ptr(), curr, next)
10:        prev = nullptr, curr = root, next = nullptr
11:        if curr.ptr() = nullptr then
12:            return {prev, curr, next, HEAD_IS_NULL}
13:        end if
14:        continue
15:    end if
16:    is_boundary = curr.markings() & BOUNDARY ≠ 0
17:    if is_boundary and curr.ptr()→key = k then
18:        return {prev, curr, next, FOUND_KEY}
19:    else if curr.ptr()→key ≥ k then
20:        return {prev, curr, next, GT_KEY}
21:    else if next.ptr() = nullptr then
22:        return {prev, curr, next, AT_END}
23:    end if
24:    prev = curr
25:    curr = next
26: end while
```

However, these boundary-marker nodes should not be included in the range since they are only metadata. Algorithmically, this is not different from skipping logically deleted nodes (i.e., nodes marked as deleted but not yet physically unlinked) as in the vCAS Harris linked list implementation [184].

Let us now consider an example of a range query traversing the data layer in Figure 6.3. A range query begins by reading the global timestamp to create a logical snapshot. If the range query reads t_0, it will only traverse the boundary-marker, finding an empty bucket, which reflects its state at time "0". If it reads t_1, it will traverse the boundary-marker and element "1". At t_2, it will traverse the boundary-marker, element "0" and element "1." At t_3, it will traverse the boundary-marker, elements "0", and find element "1" deleted. Finally, at t_4, it will traverse the boundary-marker and element "0" and then stop.

Algorithm 24 Lock Free Linked boundary-marker Insert

```
1: Given: Key k, Value v
2: while true do
3:     prev, curr, next, result = search_for_boundary(k)
4:     if result = HEAD_IS_NULL then
5:         node = new node(k, v, nullptr — BOUNDARY)
6:         if vcas(root.ptr(), nullptr, node then
7:             return true
8:         end if
9:     else if result = FOUND_KEY then
10:        return false
11:    else if result = GT_KEY then
12:        tocas = prev == nullptr ? root : prev→next
13:        node = new node(k, v, curr.ptr() — BOUNDARY)
14:        node = node — tocas.markings()
15:        if vcas(tocas.ptr(), curr, node) then
16:            return true
17:        end if
18:    else if result = AT_END then
19:        node = new node(k, v, nullptr — BOUNDARY)
20:        node = node — curr→next.markings()
21:        if vcas(curr→next.ptr(), next, node then
22:            return true
23:        end if
24:    end if
25:    continue
26: end while
```

6.3 Supporting Split and Resize

LIT's lookup functions must consider both the range of the expected key space and the number of buckets. If LIT cannot adapt to the incoming workload and account for deviations from these parameters, a change in workload can easily unbalance the data structure, resulting in a significant performance slowdown. Recall that, as opposed to hash tables, LIT does not rely on a hashing function; hence, a balanced number of elements per bucket is not guaranteed. In order to prevent these issues, LIT supports two operations: `SPLIT` and `RESIZE`.

`SPLIT` takes an index of a level (i.e., a bucket) and introduces another level above it, splitting the managed range of keys across some number of buckets. For example, Figure 6.2b Bucket 1 is split into another level of size 2. If the bucket contains more than a

Algorithm 25 Lock Free Linked Remove

```
1: Given: Key k
2: while true do
3:     prev, curr, next, result = search(k)
4:     if result = FOUND_KEY then
5:         deleted_next = next.ptr() — DELETED
6:         if vcas(curr→next.ptr(), next, deleted_next) then
7:             tocas = (prev = nullptr) ? root : prev→next
8:             vcas(tocas.ptr(), curr, next.ptr() — tocas.markings())
9:             return true
10:        end if
11:    else
12:        return false
13:    end if
14:    continue
15: end while
```

Algorithm 26 Lock Free Linked Get

```
1: Given: Key k
2: prev, curr, next, result = search(k)
3: if result = FOUND_KEY then
4:     return curr.ptr()→value
5: else
6:     return none
7: end if
```

threshold, T_n, number of nodes, LIT splits to reduce the length of an operation's traversal phase.

Over time, splitting buckets can lead to a structure similar to an unbalanced tree. That would nullify the performance benefits of LIT and its fast lookup function. For this reason, LIT supports a `RESIZE` operation, which modifies LIT's index, flattening its multiple levels and returning to the original single-level design. `RESIZE` takes in a new key range (min and max) and a new size (i.e., number of buckets) and creates a new lookup function and root level. LIT resizes to a larger root level after creating more than a predefined (T_l) number of levels.

In practice, the `RESIZE` operation is performed by inserting new boundary-marker nodes for each bucket of the new flatted index into LIT's data layer. For example, in Figure 6.4, we resize the LIT shown in Figure 6.4a into Figure 6.4b. We have resized the LIT with $N = 5$, $a = 0$, and $b = 9$ replacing the previous $N = 4$, $b = 7$, $a = 0$. It is worth noting

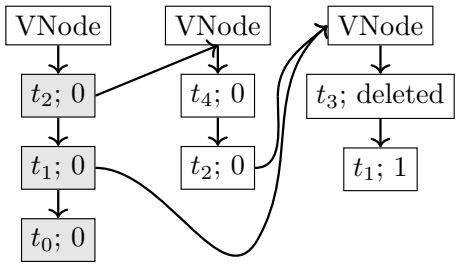


Figure 6.3: vCAS LIT Data Layer (Linked List); boundary-markers are shaded gray; pointers to null not shown; each node contains the timestamp and the value.

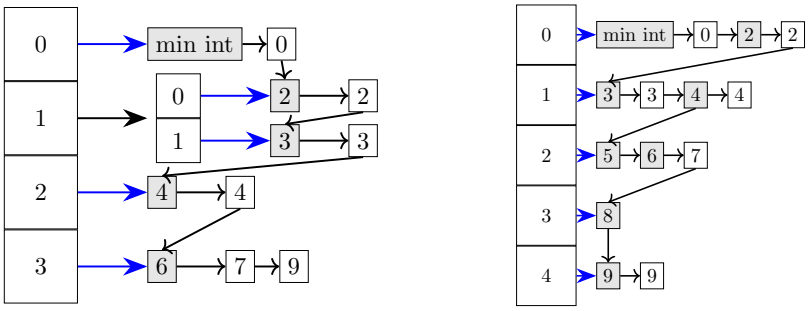
Algorithm 27 LIT Range Query

```
1: Given : keys start and end
2: results = { }
3: linkedlist = bucket_lookup(start)
4: time = Camera→snapshot()                          ▷ Take a vCAS logical snapshot
5: node = linkedlist→root.load(time).ptr()           ▷ load the root at timestamp time
6: while node != nullptr and node→key ≤ end do
7:     bitset = node→next.markings()                 ▷ Get the bitset we steal from the pointer
8:     deleted = bitset & DELETED ≠ 0
9:     boundary = bitset & BOUNDARY ≠ 0
10:    if !deleted and !boundary and node→key ≥ start then
11:        results.append((node→key, node→value))
12:    end if
13:    node = node→next.ptr()
14: end while
15: return results
```

that the choice of the `SPLIT`'s or `RESIZE`'s parameters does not affect LIT's correctness but only its performance. In our implementations, `SPLIT` and `RESIZE` are invoked based on a deterministic heuristic, which we discuss below.

The boundary-markers associated with the previous structure are left in the data layer to allow concurrent bucket lookups to operate without blocking conditions due to resizing. Once operations can no longer see the previous index layer (i.e., the oldest concurrent operation has a timestamp greater than the timestamp of the most recent boundary-marker in the structure before `RESIZE`), these former boundary-markers can be physically removed from the data layer.

Two important considerations should be made regarding the `RESIZE` and `SPLIT` operations. First, boundary-markers are versioned through vCAS. Therefore, LIT's operations



(a) Before Resize (b) After Resize Returns

Figure 6.4: Resizing LIT; enabling flattening of levels

can proceed concurrently with modifications to the index layer and without interference. Only operations started after the completion of `RESIZE` or `SPLIT` will observe the new boundary-markers and, therefore, the new structure. Second, as opposed to the resize operation of hash table data structures, in LIT, a resize or split must preserve the ordering of keys to serve range queries correctly. That means, given two keys in the data layer, regardless of which buckets they belong to, their order will always be the same on the data layer after either `SPLIT` or `RESIZE`.

The following details how we use versioning to enable concurrent modifications of the index and data layers.

6.3.1 SPLIT

`SPLIT` is implemented as Algorithm 28. If the algorithm returns true, it means that the thread successfully split a bucket. If the algorithm returns false, it means that another thread split the bucket. Since inserting boundary-markers is lock-free, this is lock-free as well.

We begin a `SPLIT` by getting a level and a bucket, as well as a size into which we intend to split the bucket. We then calculate the minimum and maximum keys that map the current lookup function to the level bucket. After this, we use these parameters to compute a new lookup function (Line 7), insert new boundary-markers (if needed), and create a new level with pointers to these boundary-markers. After this, we can perform our vCAS to attempt to modify the bucket to point to this new level rather than the data layer.

Algorithm 28 Split

```
1: Given: level L, bucket i, and size N
2: ll = L[i]                          ▷ L[i] must be a linked list for split to be called
3: domain = L→inversef(i)                                          ▷ Returns domain of f
4: a = domain.min                                  ▷ Minimum key that f maps to bucket i
5: b = domain.max                                    ▷ Note: the domain does not contain b
6: c = N/(b-a)
7: f'(x) = clamp(⌊c(x − a)⌋, 0, N − 1) ▷ Bound new lookup function between 0 and N − 1
8: newlevel = new level(f'(x))
9: for i = 0; i < N; i++ do
10:     k = inversef'(i).min
11:     newlevel[i] = ll→insert_or_get_boundary_marker(k)
12: end for
13: if vCAS(L[i], ll, newlevel) then                          ▷ CAS in the new level with vCAS
14:     return true
15: end if
16: return false
```

In practice, to avoid tracking the number of elements in each bucket, we only split during insert operations and only if a certain number of traversals of the data layer are observed to get to the key. We set this threshold to 10. We split the bucket into another level with multiple buckets (in our implementation, 10 or the distance between the minimum and maximum keys that the bucket indexes divided by 10). This process can cause recursive splits depending on the key universe. However, recursive splits lead to resizing, which in practice improves performance and can lead to reaching a stable state for finite key spaces.

Our heuristic is deterministic and consistent across multiple threads trying to split the same bucket. Every thread performing a split on a given bucket will compute the same new lookup function. This means that we will also split it to a consistent size. Therefore, if a split fails to change the level, another split with the same parameters succeeds. Therefore, a split never leaves boundary-markers that are unused (i.e., mapped to by the index layer).

6.3.2 RESIZE

`RESIZE` is implemented as Algorithm 29. If the `RESIZE` returns true, it means that our thread updated the first level. If it returns false, it means another thread did. To resize, we create a new lookup function (at Line 4), insert new boundary-markers (if necessary), and then attempt a versioned CAS on the root level. In practice, similar to `SPLIT`, we use

a heuristic checked during insert operations. If we traverse too many levels due to prior splits, we will resize to limit the amount of splitting. In our implementation, we split after traversing 10 or more index layers before reaching the data layer.

Algorithm 29 Resize

```
1: Given: Size N, min key a, and max key b, expected root level L
2: ll = bucket_lookup(minimum representable key)
3: c = N/(b-a)
4: f(x) = clamp(⌊c(x − a)⌋, 0, N − 1)
5: newlevel = new level(f(x))
6: for i = 0; i < N; i++ do
7:     k = inverse f(i).min                          ▷ Get minimum key that maps to i
8:     newlevel[i] = ll→insert_or_get_boundary_marker(k)
9: end for
10: if vCAS(root_level, L, newlevel) then
11:     return true
12: end if
13: return false
```

Unlike `SPLIT`, `RESIZE` can leave boundary-markers not mapped by the index layer. Also, it is possible that multiple `RESIZE`s attempt to execute concurrently with different new lookup functions. Although possible, only one can succeed. The unused boundary-markers can be removed after all operations concurrent with the `RESIZE` complete.

6.3.3 Bucket Lookup

In order to support both `SPLIT` and `RESIZE` while performing concurrent modifications on the data layer, we rely on the invariant that when a bucket points to the data layer (a linked list), it always points to a boundary-marker in the linked list. Furthermore, this boundary-marker's key is the minimum key that maps to that bucket (i.e., $\text{argmin}_k g(k) = i$ for bucket i and function g).

Since we can always traverse to a boundary-marker preceding the key we are interested in reaching, we can modify the LIT bucket lookup algorithm (Algorithm 20) to utilize vCAS's snapshotting mechanism to traverse a consistent state of the index layer. The resulting pseudocode of the new LIT bucket lookup algorithm that accounts for concurrent modifications is in Algorithm 30.

Since our `SPLIT`s and `RESIZE`s only need to modify the index layer with a single vCAS

Algorithm 30 Bucket Lookup with Concurrent Index Layer Modifications

```
1: Given: Key k
2: time = Camera→snapshot()   ▷ We now will read the index layer at a logical snapshot
3: level = root.load(time)
4: idx = level→f(k)
5: while level[idx] is not a linked list do
6:     level = level[idx].load(time)
7:     idx = level→f(k)
8: end while
9: return level[idx].load(time)
```

operation, data structure operations can only observe a composition of the data layer that is consistent either with the state before a `SPLIT`/`RESIZE` occurs or after. For correctness' sake, the index layer could be traversed with any arbitrarily old timestamp, and its correctness would still be ensured. This is because the traversal will surely reach a boundary-marker ordered before the node of interest.

On the other hand, once an update operation returns from the bucket lookup function and starts traversing the data layer, it would be incorrect to traverse the data layer using an old timestamp. LIT solves this issue by rereading the timestamp when moving from the index to the data layer. This is commonly done in data structures with probabilistic indexes (e.g., bundled skip lists [130]), and linearizes operations at or after the second read of the timestamp.

6.3.4 Performance Analysis

In the absence of `RESIZE` and `SPLIT` operations, the worst-case performance of LIT's elementwise operations is bounded to $O(T_n + T_l)$, where T_n is the threshold for the maximum number of nodes per bucket before a split and T_l is the maximum number of levels before a split. In fact, with less than T_l levels and T_n nodes, an operation must traverse at most T_l levels to get to the data layer and $T_n + 1$ nodes to get to a key. Each `SPLIT` and `RESIZE` enables us to keep this bound. In practice, each `RESIZE` minimizes the need for future `SPLIT`s and `RESIZE` as long as the distance function correctly estimates the number of elements within a range and reflects the input distribution.

Assuming a finite universe of keys and a lookup function with the properties described

above, regardless of the input's probability distribution, LIT's structure will reach a stable state, meaning no more `SPLIT` and `RESIZE` operations invoked when the number of buckets is greater than or equal to the cardinality of the key space. With specific input distributions (e.g., uniform or Zipfian), we observe that LIT's structure reaches its stable state soon. To favor reaching this stable state, upon splitting, we subdivide the buckets so that it prevents more than T_n keys from mapping to any bucket that is not either the first or the last (which may hold elements less than the expected minimum or greater than the expected maximum). LIT does this by splitting each bucket into $|\{a, ..., b\}|/T_n$ buckets, where a is the minimum key and b is the maximum key the bucket can contain. This method of splitting prevents further splits of interior buckets. To further adapt to keys that are inserted outside the current indexed range (i.e., the range of level 0), we perform a `RESIZE` with the minimum and maximum keys that are present in the data structure. This enables the index to adapt to the workload as new keys are added.

Upon converging to a stable state, LIT can have at most two levels for each of the interior buckets (i.e., all buckets except the first and the last bucket). The first and the last buckets, however, can have up to T_l levels. Each of these levels can have nodes on the order of the size of the universe of keys. However, since it can happen only for first and for the last bucket, the size of LIT is linear in the cardinality of the universe, in the worst case.

6.4 Adapting LIT for Heterogeneity

The traditional disconnection between CPU-based data structure designs and GPUs is mostly attributed to the inherent linked structure of memory, which entails long sessions of pointer chasing to accomplish operations. This way of performing traversals does not meet the GPU architecture/runtime requirements because of the absence of a lock-step structure in the operation. On the other hand, LIT limits the length of traversals by exploiting a hash table-like index layer, which shows more affinity to the GPU requirement. This is confirmed by existing high-performance hash table implementations in literature (e.g., SlabHash [18]). For this reason, we believe a GPU-based LIT design is beneficial. Indeed, it is possible to directly port and execute a data structure implementation on modern GPUs

using frameworks like the CUDA C++ Core Libraries [36]. However, without programming with GPU-specific optimizations, performance will be poor.

In this section, we describe how we modified LIT's design to be executed on a heterogeneous device, such as the GPU. We start by quickly summarizing the typical GPU programming model, which is presented by the parallel thread execution (PTX) instruction set architecture and consists of cooperative thread arrays (CTAs). CTAs are made up of threads that are launched on the same streaming multiprocessor. To execute computation on a GPU, a grid of these CTAs is launched. The threads operate in a single instruction multiple thread (SIMT) model in which each thread within a group of 32 threads (warp) typically executes the same instruction but on different data. Coordination between the threads may be necessary and can be optimized. Many optimizations exist for groupings of threads within the CTA; for example, warps can coalesce memory accesses for performance, broadcast data to its threads, and perform other coordination instructions.

To adapt LIT to the GPU, we build upon CODS presented in Chapter 5. CODS is a uniform framework for designing data structures with coalesced memory access optimizations. In the context of GPUs, CODS presents an object, called a c-object, that we can utilize to read and write to memory through `get`, `set`, and elementwise atomic operations. This object natively supports the warp coalesced optimization. For LIT, we extend CODs to support vCAS versioning on the entire c-object.

The primary change to support vCAS versioning is to ensure any `set` or atomic operation on an index results in a vCAS CAS of the c-object. Similarly, `get` or atomic elementwise operations must read the c-object with vCAS either at a specific version or as the latest vNode.

With these modifications, we create a new linked list that supports warp cooperative optimization. This GPU-aware linked list uses multi-element nodes of 32 key-value pairs rather than single-element nodes, as in the case of the CPU. We store all keys and values in contiguous memory. This enables us to coalesce access to the keys when searching for a key-value pair, and once found, the value can be accessed if necessary.

Porting vCAS to GPU. We begin our porting by making our vCAS functions safe to execute across warps. The first issue to deal with is that, because of concurrency, if all

Algorithm 31 GPU vCAS Load

```
1: Given: vCAS object O
2: value = nullptr
3: if is_loader_thread() then
4:     value = O.load()
5: end if
6: return broadcast(value, loader_id())
```

Algorithm 32 GPU vCAS

```
1: Given: vCAS object O, expected E, desired D
2: result = false
3: if is_leader_thread() then
4:     result = O.cas(E, D)
5: end if
6: return broadcast(result, leader_id())
```

threads of warp access the vCAS object, they might read different versions of it. Therefore, to correctly read a vCAS object in a warp, we assign the role of "loader" to one thread of the warp. The loader thread performs the read and then shares the result with the rest of the warp (Algorithm 31). In LIT, since we do not update key-value pairs or lookup functions in place, any non-vCAS objects are immutable and can be read without passing through the loader thread. This enables us to safely read c-objects after reading from our vCAS objects. Our modified vCAS operation for GPU warp execution is shown in Algorithm 32. Similarly, we assign the first thread in the warp the role of "leader" and perform the operation. vCAS objects also provide a third API to read a version given a specific timestamp. That API is already safe for a warp to perform. Since prior versions cannot be modified, any thread that attempts to read the same version will return the same value.

Utilizing Multi-element Nodes. To achieve optimal performance on the GPU, we need to enable support for multi-element nodes and coalesce accesses. This means that we should access elements of the same cache lines to reduce memory transactions from our warp. To accomplish this, each thread in the warp must be assigned to a distinct key-value pair in the node. When searching for a key and accessing the node, we use the warp to collectively load all keys in the node, and compare them in parallel with the key(s) we are looking for. The warp then collects the search results from each of the threads and aggregates them. These results contain information about whether the key was found or

other information, such as whether the entire warp contains keys greater than the key(s) being searched for. Within a multi-element node, we do not order the keys; instead, we have the warp determine the minimum and maximum keys while searching.

An important effect of introducing multi-element nodes is that, with this structure, upon insert or remove operations, threads should now be able to change the content of this multi-element node, as opposed to creating a new node (for `INSERT`s) or operating on pointers (for `REMOVE`s) in the CPU version. Unfortunately, these modifications are not trivially compatible with vCAS. To fix this issue, we introduce a vCAS-aware locking mechanism for multi-element nodes. This means that while the CPU has lock-free operations and a wait-free range query, the GPU has a lock-free get, a wait-free range query, and blocking insert and remove.

Algorithm 33 Multi-element Node Linked List Search

```
1: Given: Key k
2: prev = nullptr, curr = root.load(), next = nullptr
3: if curr.ptr() = nullptr then
4:     return {prev, curr, next, HEAD_IS_NULL}
5: end if
6: while true do
7:     next = curr→next.load()
8:     if curr.markings() & DELETED then
9:         if not curr.markings() & LOCKED then
10:            vCAS(prev→next.ptr(), curr, next)
11:        end if
12:        prev = nullptr, curr = root.load(), next = nullptr
13:        if curr.ptr() = nullptr then
14:            return {prev, curr, next, HEAD_IS_NULL}
15:        end if
16:        continue
17:    end if
```

vCAS-aware locking mechanism. Our vCAS-aware locking mechanism implements locks by stealing a third bit from the pointer to the next node and still utilizes vCAS to modify the data structure. Similar to before, we keep our boundary-markers as a node with a single key and do not use multi-element nodes. This enables us to support the invariants necessary for splitting, resizing, and searching through the index layer. Unlike the CPU's LIT, any update operation is now required to copy and modify the multi-elements node and

link it to the new version created using vCAS. Also, since we utilize a vCAS-aware lock and modify internal node data by using vCAS, concurrent threads can correctly continue to perform non-blocking traversals without checking if the visited node is locked.

```
18:     boundary = curr.markings() & BOUNDARY ≠ 0
19:     min = curr.ptr()→min()
20:     max = curr.ptr()→max()
21:     idx = curr.ptr()→contains(k)
22:     if !boundary and idx ≠ none then
23:         return {prev, curr, next, idx, FOUND_KEY}
24:     else if k > min and k < max then
25:         return {prev, curr, next, IN_RANGE}
26:     else if min > k or (boundary and min ≥ k) then
27:         return {prev, curr, next, GT_KEY}
28:     else if next.ptr() = nullptr then
29:         return {prev, curr, next, AT_END}
30:     end if
31:     prev = curr
32:     curr = next
33: end while
```

To support operations on multi-element nodes, we modify our data layer search (Algorithm 33) by introducing a new return condition to inform insert operations of which multi-element node should be targeted to finalize the insert.

For `GET` or `RANGE_QUERY`, we only need to read the key-value pairs that match the key or range we are looking for, respectively, and do not need to acquire locks. Below, we focus on `INSERT` and `REMOVE` operations.

To perform `INSERT`, if the key is already present, we return false. If not, we should check if there is enough room in the multi-element node to store that key. If so, we lock the previous and the current node and then create a copy of the current node with the new key-value pair inserted. Then, we update, though vCAS, the next pointer of the previous node to the just created multi-element node and finally unlock. On the other hand, if there is not enough room, or we are inserting a boundary-marker, we must split the node into two. To split a node, we lock the previous and the current, then find all the keys less than our key and all the keys greater than our key and create a node for each set. We can insert our key into one of the nodes or between if it is a boundary-marker. Then, we can proceed with updating the pointer to the current through vCAS.

To implement `REMOVE`, if we find the key and the node has only one element in it (our key), we can use a vCAS to mark the node as deleted. In the case the node has more than one element, we must lock the previous and the current nodes and update the current node with a copy of it with the key removed.

During `INSERT` or `REMOVE`, if we find that any node that we are operating on is locked, we must retry our operation. By doing this, we are able to guarantee that there are no lost updates.

6.5 Evaluation

To assess the performance of LIT, we contrast its performance against state-of-the-art logarithmic linked data structures that support highly efficient range queries: a skip list, enhanced with the vCAS [184] and the bundling [130] versioning techniques, and a vCAS-enabled CITRUS tree set [15]. Since LIT is inspired by the structure of a chaining hash table, we also compare it with Michael's hash table [118] for elemental operations as an upper bound on performance. While LIT must maintain ordering and versioning, the hash table does not order keys to support range queries and, therefore, can perform significantly faster than any other competitor.

We compare these approaches by utilizing the existing implementations from [130] for the skip lists and a modified synchrobench [77] for Michael's hash table. In our evaluation, we consider a closed-loop workload with uniform and skewed (zipfian) access patterns. When evaluating range queries, we fix the ranges to 50 keys. We run each experiment with a billion operations five times and then report the average performance. We disable memory reclamation for all competitors.

We utilize a testbed with 4 Intel Xeon Platinum 8160 CPU with hyper-threading enabled, giving us 96 cores with 192 hardware threads. Unless otherwise stated, we evaluate by running on all 192 threads. We run with 8B keys and 8B values. Our implementation is in C++, built with the clang++-15 compiler.

We further evaluate LIT by extending it to the GPU and evaluating it in comparison to GPU B-trees [23, 21] on an NVIDIA RTX 4090. The B-trees are specifically designed

for indexing 4B key and 4B value pairs by utilizing 8B atomic operations, as well as having node sizes at the L1 cache granularity (128B) and coalescing accesses to the nodes. This provides a highly optimized competitor.

We begin by evaluating LIT's CPU performance by changing the key range over a variety of uniform workloads. We prepopulate the data structures with half of the key range and the number of buckets for LIT to 10k.

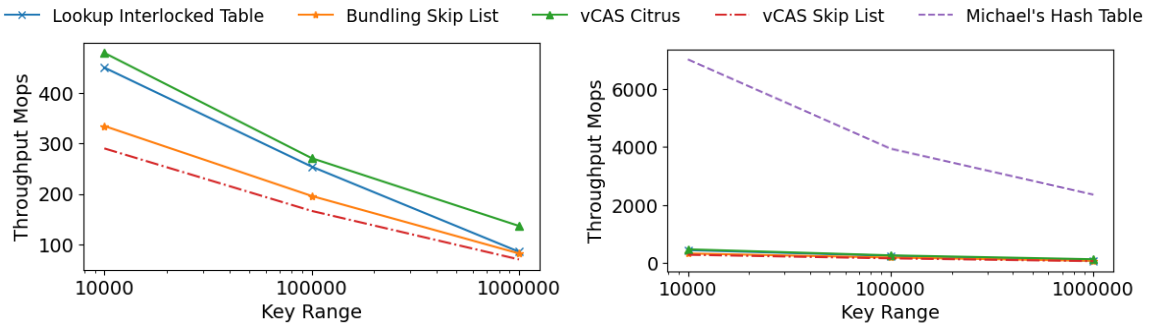


Figure 6.5: Uniform 100% Gets While Varying Key Range.

In Figure 6.5, we consider how performance changes as the key range increases for get operations. LIT outperforms the skip list data structures at small key ranges by 1.3x over a bundled skip list and 1.5x over a vCAS skip list and continues to outperform at higher key ranges with a less significant performance gap. LIT is 4% faster than a bundled skip list and 21.4% faster than vCAS skip list at a key range of 1M. This is largely due to the fewer traversals LIT must perform. In fact, for these operations, we find that for 10k keys, we traverse one level and 1.03 nodes on average to reach the desired key. For 100k keys, we traverse one level and 3.7 nodes. At 1M, we traverse two levels on average and 3.2 nodes. With smaller traversals and key ranges, LIT is able to cache accesses more effectively than the skip lists, which gives it a competitive edge. Compared to the CITRUS tree, LIT is up to 37% slower. This performance advantage is due to the fact that in read-only workloads, the tree does not need to traverse boundary markers like LIT and does not need to read values since it is a set.

Michael's hash table is able to achieve significantly higher performance over all other competitors. This is due to the map not needing to check versions. Without these constraints, Michael's hash table is able to efficiently hash to a bucket and traverse the ordered

list used for the bucket until it finds the key. Also, `RESIZE`s never occur, contributing to its high performance.

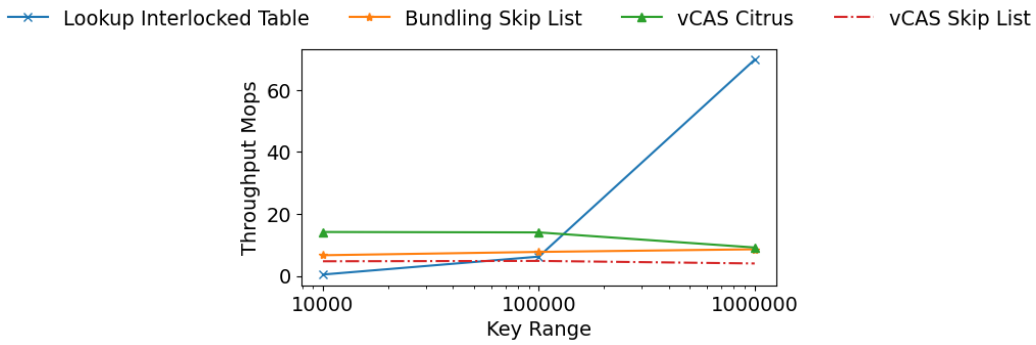


Figure 6.6: Uniform 50% Inserts and 50% Removes While Varying Key Range.

When we consider the data structures in an update-only workload with an even mix of `INSERT`s and `REMOVE`s in Figure 6.6, we find that LIT is able to outperform the skip lists when increasing the key ranges. At 10k keys, LIT is slower than vCAS and a bundled skip list. This can be attributed to the pre-fetching effect of traversing the linked indexes of the skip lists. In fact, in high contention, cache invalidations dominate the performance. While all data structures see these effects, due to the common traversal path of such a small key range in the skip lists, it is constantly updating its cache with elements that may be accessed in the future. When we run the experiments with the `perf` tool to collect statistics, we see that 0.5% of all L1 data cache loads are misses for a vCAS skip list, but for LIT this is 25% of all L1 data cache accesses.

As we increase the key range, the percentage of L1 data cache misses for LIT decreases to 4% of all L1 data cache accesses. At 100k keys, LIT has a 1.3x speedup over a vCAS skip list and a within 20% of the performance of a bundled skip list. At 1M keys, LIT is able to achieve a 17.4x speedup over a vCAS skip list and a 8.1x performance improvement over a bundled skip list. While skip lists must track the prior node seen on each index level and will contend on the index when locking and modifying the underlying key-value pairs, LIT does not. This enables LIT to scale, especially as contention decreases (e.g., with a larger key range). The CITRUS tree outperforms the skip lists; however, its performance is stagnant across key range sizes. At 1M keys, LIT is able to achieve a 7.6x speedup over the CITRUS tree.

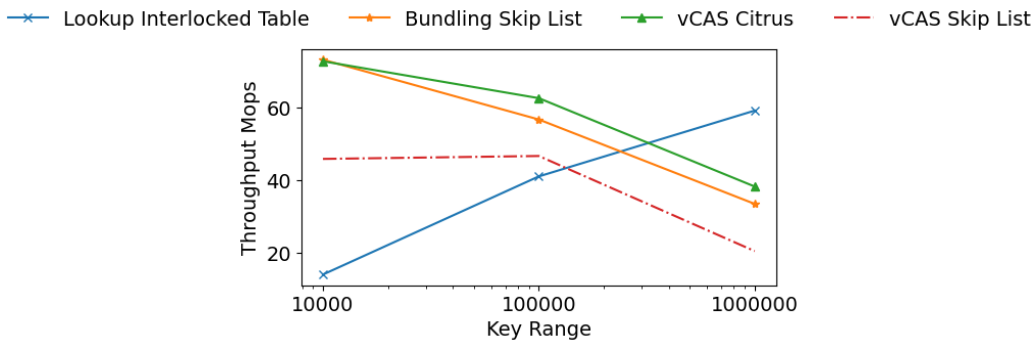


Figure 6.7: Uniform 10% Range Queries, 10% Updates, 80% Gets While Varying Key Range.

In Figure 6.7, we consider a mixed workload with 10% range queries, 10% updates, and 80% gets. Consistently with what we observed earlier, LIT outperforms a versioned skip list at large key ranges. Initially, LIT only achieves 59% of the performance of a bundled skip list. This is due to the same caching effects as described for 50% `INSERT`s and 50% `REMOVE`s. As the key range increases to 1M, LIT outperforms the vCAS skip list by 2.9x and the bundled skip list by 1.8x. Compared to the CITRUS tree, LIT outperforms it by 1.5x at 1M keys. This is due to LIT's ability to better handle `INSERT`s and `REMOVE`s at these large key ranges. Even at 10% modifications, the skip lists and CITRUS tree have issues maintaining performance.

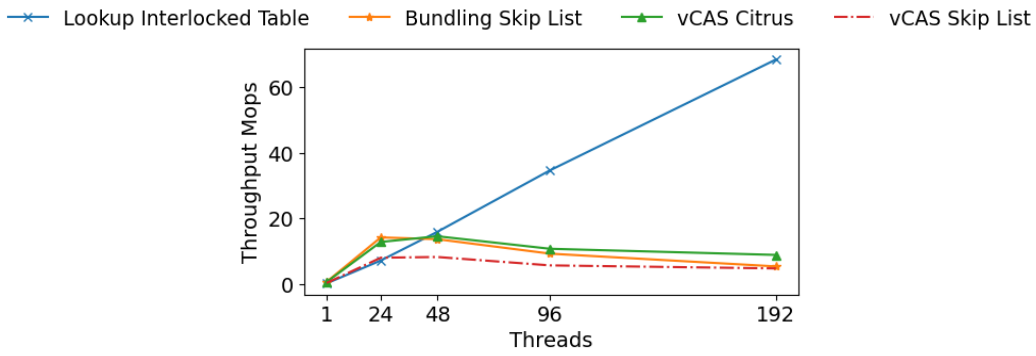


Figure 6.8: Scaling During Modifications.

Next, we conducted experiments to observe scalability by increasing the number of threads in the system. For brevity, we omit cases where all competitors scale and instead focus on 50% `INSERT`s and 50% `REMOVE`s with a key range of 1M elements and an initial population of 500k keys. In Figure 6.8, we plot results. While LIT is able to scale with

increasing threads, the skip lists do not. Beyond 24 threads, all competitors begin executing across non-uniform memory access (NUMA) zones, and at 96 threads, we execute across all four NUMA zones; then, it is hyperthreading. At 48 threads, LIT achieves 1.9x the throughput of a vCAS skip list, 1.2x the throughput of a bundled skip list, and a 9% increase in throughput over the CITRUS tree. When we reach 192 threads, LIT achieves significant speedups against all competitors, as also observed in Figure 6.6, which shows the same experiment with 192 threads. LIT is 12.5x the throughput of the bundled skip list, 14.1x the throughput of the vCAS skip list, and 7.6x the throughput of the CITRUS tree.

Due to the common traversal paths of all competitors, performing CAS operations on the index causes cache invalidations, which must cross NUMA zones. LIT, however, mainly modifies the underlying linked list, which will only cause conflicts if another thread is concurrently searching through the same section of the list or trying to modify the same node. This scalability trend is similar to what we expect in chaining hash tables, where conflicts between threads updating key-value pairs would only occur if two keys hash to the same bucket.

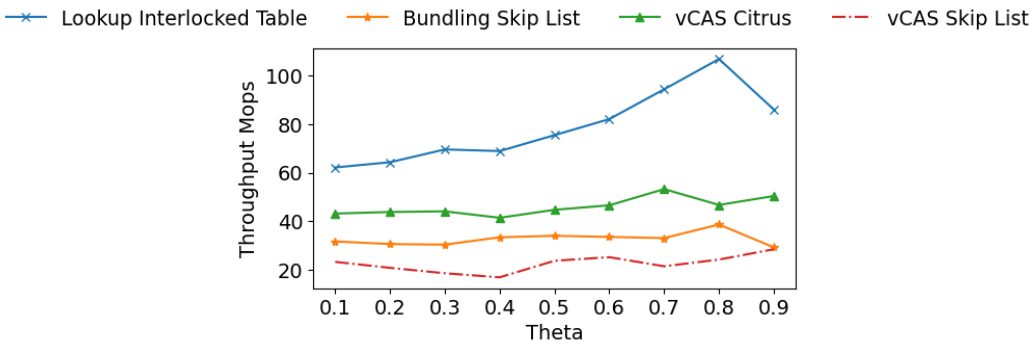


Figure 6.9: Zipfian 10% Range Queries, 10% Updates, 80% Gets; Key Range 1M; Varying Theta.

In Figure 6.9, we evaluate the mixed workload with a key range of 1M and an initial population of 500k while varying a zipfian distribution ordered along the keys (i.e., accessing key "1" has a higher probability than accessing key "2", and accessing key "2" has a higher probability than 3, and so on). At a theta of 0.1, LIT has 2.7x the throughput of a vCAS skip list, 2.0x the throughput of a bundled skip list, and 1.4x the throughput of the CITRUS tree. As we increase theta to 0.8, the performance of LIT increases due to the

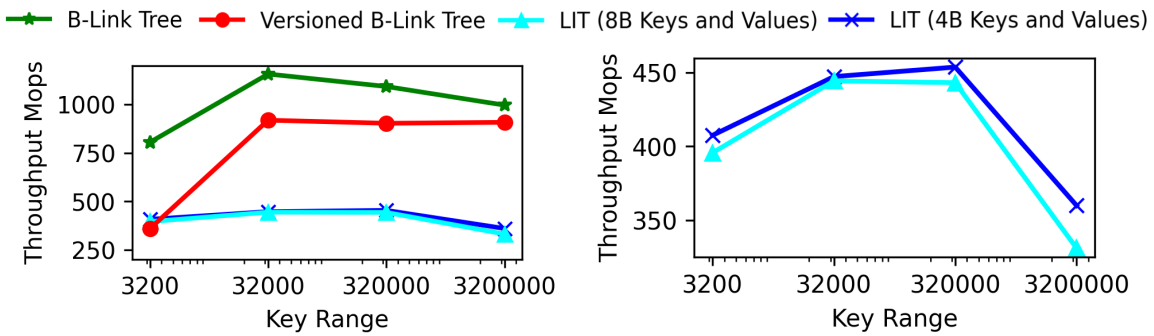


Figure 6.10: Varying Ranges of a 80% Get, 10% Update, and 10% Range Query Uniform Workload on the GPU.

skewed nature of accesses, meaning that LIT will have increasing locality and few traversals enabling caching. This leads to LIT outperforming the vCAS skip list by 4.4x, bundling by 2.8x, and the tree by 2.2x.

At a theta of 0.9, however, LIT begins to conflict on only a few elements. However, this does not introduce any splits or resizes. Since we initially prepopulate the data structure with 500k keys with a bucket size of 10k, we expect LIT to split during prepopulation to a second level for each bucket without any need for future splits or resizes. Rather than degrading performance due to structural operations (e.g., split and resize), LIT degrades performance because of the logical contention of operations.

Table 6.1: Breakdown of Performance; 1M Key Range; 100% Inserts.

Optimization	Throughput
No Splits or Resize	26.7 Mops
Only Resize	31 Mops
Splits and Resize	31.1 Mops

To further understand the impact of splitting and resizing, we also evaluate an insert-only workload with a key range of 1M, an initial population of 500k keys, and an initial level size of 10k. Results are shown in Table 6.1. When we run with `SPLIT`s and `RESIZE`s disabled, we see that LIT is only able to achieve 26.7 Mops, but enabling `RESIZE` from this point increases performance by 16%. Furthermore, enabling splitting can continue to increase performance but by only 0.6%.

We evaluate the performance of the GPU version of LIT in Figure 6.10. Here, we

execute 1M operations of our 80% get, 10% update, 10% range query workload mix across a variety of key ranges. Our competitors are B-link tree [21] and Versioned B-link tree [23]. Both of them are data structures designed specifically for the GPU. Importantly, to retain correctness, the design of these trees requires that keys and values have a size of 4 bytes. Unlike them, LIT is not limited to 4B for its keys and values. Accordingly, we execute LIT with two configurations: the first uses 4B keys and 4B values like the competitors, and the second uses 8B keys and 8B values. We set the size of LIT for each execution to be 1/10th of the range for optimal performance on the GPU. For all competitors, we execute batches of operations in concurrent CUDA streams. Performance is measured from kernel launch to synchronizing the streams. In other words, CPU/GPU data transfer is not included in the performance, hence the whole memory is assumed to be resident on the GPU. We find that LIT is able to reach up to 454 Mops of throughput with 4B keys and values, and only degrades by a maximum of 8% when moving to larger keys. In comparison to data structures specifically optimized for the GPU with 4B keys and 4B values, we see B-Link tree performs the best with up to 2.8x the speedup over LIT, while applying versioning to the tree similar to LIT is up to 2.5x the throughput. This performance difference is expected since although all data structures are optimized to coalesce accesses, competitors do not have to perform the initial VNode dereference as LIT. This optimization is specific to the 4B key and 4B value design and would not be portable to larger key-value sizes. We, however, believe this is a fair tradeoff to be applicable for larger key and value sizes.

6.6 Related Work

Range query data structures have been of recent interest [16, 184, 130, 23, 21, 197, 25, 40, 20] due to the applicability to data repositories and the complexity that such a problem provides when supporting linearizability. Many modern data repositories utilize range queries, such as RocksDB [57], which utilizes a log-structured merge tree. Many modern data repositories also make use of GPUs, such as RateupDB [107], which utilizes the GPU and CPU to store data for transactional and analytical processing. Key-value stores, such as MegaKV [192] and KVCG [53], utilize the GPU to store key-value pairs which are operated on with

linearizable operations.

We build on prior work on adapting data structures to support range queries. Four recent works, EBR-RQ [16], and Bundling [130], versioned CAS [184] (vCAS), and Verlib [29] all enable supporting range queries within existing data structures. EBR-RQ relies on the epoch-based reclamation mechanism utilized by many data structures to perform range queries by iterating through both the data structure and the set of retired objects. Bundling introduces the concept of bundled entries, which enables the versioning of the data structure with a locking mechanism. Versioned CAS, utilized in our implementation of LIT, creates versioned objects that can be operated on with atomic reads and versioned reads and CAS operations. This enables the transformation of lock-free data structures to support wait-free range queries. Verlib generalizes Bundling and vCAS by considering various timestamping schemes (on CAS or snapshot) and considers the necessity of always timestamping to linearize range queries. However, the techniques Verlib generalized were already applicable to LIT and other list-based data structures.

LIT is inspired by the design of hash tables, which utilize hash functions to index data in an unordered fashion. Chaining hash tables, such as Michael's hash table [118], utilize both linked lists and an index to provide lock-free operations, similar to LIT. Unlike LIT, these kinds of hash tables cannot provide range queries. Since these hash tables do not have an ordering like a lookup table, supporting range queries would require traversing the entire data set.

LIT is also inspired by the interlocking hashed table [96] (IHT), which can have multiple levels of indexes to map to data. Unlike IHT, the LIT is a lock-free data structure and provides ordering over keys to enable support for range queries.

The SlabHash [18] and L-Slab [53] GPU hash tables also inspire the design of the GPU version of LIT. Similar to both these data structures, LIT makes use of warp cooperation, where 32 GPU threads collectively work to perform an operation. Unlike Slab, LIT is not wholly lock-free on the GPU and instead must block on `INSERT`s and `REMOVE`s to retain ordering constraints. Unlike both SlabHash and L-Slab, LIT supports range queries with wait-free progress guarantees.

More closely related to LIT are data structures that can support range queries. The work

in [197] implements a log structures merge tree (LSM-tree) with support for range queries. However, we do not compare it to such a data structure since LSM-trees are designed for write-heavy workloads and typically utilize the disk for persistence because logging is very fast. LIT, however, is geared toward in-memory data structures with read-heavy workloads.

Skip lists [150, 56] are very closely related to LIT. Both enable indexing an underlying linked list to reduce the number of traversals it takes to get to the data that is being queried. Both can also utilize versioning, such as vCAS [184], to support range queries. However, unlike skip lists, LIT uses a lookup function rather than smaller lists to index the data, and it can change the number of traversals needed to access an element over time by either resizing or splitting.

On the GPU, data structures such as adaptive radix trees [6] and B-Trees [23, 21] are utilized to index data and provide range queries. Adaptive radix trees provide an efficient way to index keys with long strings of common bytes, however, our initial design of LIT does not focus on optimizing for these common bytes. B-trees, however, are efficient in ordering and indexing key-value pairs on the GPU. [23] extends [21] to a multi-versioned B-tree for performing point queries and range operations on the GPU. Unlike LIT, B-trees are less dynamic in changing the indexing because they must maintain balance, which LIT is not required to do.

6.7 Summary

In this chapter, we have introduced LIT, a mapping data structure that is inspired by the function-based indexing of a hash table. The unique feature of LIT is that it provides ordering of keys, which can be used to support range queries. After conducting experiments, we have found that LIT can achieve up to a 2.2x speedup in mixed operation workloads over state-of-the-art approaches.

Chapter 7

Conclusion

In this dissertation, we demonstrated an approach for designing data repositories and components with heterogeneous systems through the lens of instruction set architecture (ISA) affinity. With Moore's law ending and it becoming more challenging to design performant systems without relying on interconnection networks (i.e., NVLINK, QPI, Infinity Fabric, UCIe, PCIe, CXL, etc.), it is important to reevaluate what processors are utilized for modern computer system design. Our methodology has been to approach a specific application from the perspective of ISA affinity while understanding this interconnection network's additional performance cost. Through this approach, we can redesign algorithms and system architectures to achieve higher performance on more specialized processors. In these chapters, we have focused explicitly on x86_64 CPUs and GPUs.

In Chapter 2, we focused on developing a cooperative CPU-GPU-based key-value store called KVCG. In KVCG, we characterize GPU hash tables as sustaining a higher throughput than CPU hash tables and CPU hash tables, achieving lower latency. From this understanding of performance characteristics, we route requests to the CPU or GPU hash table in a key-value store to maintain desirable performance characteristics for typical skewed workloads. Furthermore, we design this routing method to maintain linearizability, which is expected of most key-value stores for programmability and correctness.

SnapKV, in Chapter 3, similarly evaluates how we can design data repositories with heterogeneity in mind. We focus on online transactional processing, online analytical processing, online decision augmentation workloads, and modern data structure approaches to

design a generic data repository that can handle these workloads efficiently. With first-class GPU support, SnapKV can accelerate computation within a transactional setting. Doing so can decrease the latency of transactions, reducing the potential for many long-running transactions to overlap and cause a high abort rate. To aid in this first-class GPU support, we introduce a new primitive `SNAPSHOT`, which leverages modern data structure support for linearizable range queries.

KVCG and SnapKV utilize the concept of ISA affinity in their approaches to addressing the performance of modern data repository systems. This understanding allows tasks to be appropriately assigned to each processor, achieving state-of-the-art performance. Furthermore, we designed both of these systems while retaining programmability for the end user.

In the following chapters, we focus on components of data repositories. In Chapter 4, we focus on designing a system for transactional processing of transactions that correspond to the semantics of sending and receiving currency. Again, we tackle this problem by breaking up parts of the computation, such as low-latency data storage and retrieval, as a CPU task and highly parallel clustering of transactions as tasks suited to GPUs. By doing so, we outperform the state of the art significantly. We also demonstrate that this approach is more practical from a performance perspective than just utilizing a CPU for enterprises and consumers.

We design the CODS framework in Chapter 5 for coalescing memory accesses in data structures and generalizing this optimization across architectures. While the work only targets CPUs, GPUs, and RDMA, this optimization is more general. CODS enables the adaptation of this optimization to more architectures to achieve high programmability. Furthermore, through CODS, we demonstrate that there are tradeoffs with these optimizations that the framework enables to be tuned and designed for. We also demonstrate that it consistently results in performance improvements for data structures.

Finally, in Chapter 6, we design a practical data structure for linearizable point operations and range queries called LIT. These data structure operations are essential to modern data repositories, including SnapKV. Furthermore, we utilize CODS to generalize LIT to the CPU and GPU. We demonstrate that LIT improves performance over trees and skip

lists on many relevant workloads and that the generalization also enables programmable GPU range queries.

CODS and LIT's approach does not focus on a single architecture. Instead, it aims for a generalizable approach so that designers may utilize generic algorithms and data structures for heterogeneous processors as they see fit through the lens of ISA affinity.

7.1 Looking Forward

This work opens avenues for further research in two distinct but related directions: generalization to other architectures and generalization to different domains.

Generalization Across Architectures. While this work has focused primarily on data repositories and heterogeneous systems through the lens of ISA affinity, we have also focused on programmability as an essential feature of heterogeneous systems. Since heterogeneous architectures necessarily mean programming in a specialized or targeted way, engineering and evaluating multiple solutions to understand ISA affinity may be infeasible. We ease the burden with highly-programmable approaches, allowing quicker iterations to characterize affinity.

Our solutions, especially CODS, can be generalized to become more programmable and accessible. While this work only considers x86_64 CPUs and GPUs, it is essential to consider other architectures and the affinity of our problems to them. Without extending these approaches, future researchers may find themselves unaware of existing solutions done in the context of a separate architecture delaying or preventing the design of optimized solutions. Chapter 5 highlights this siloed nature of CPU, GPU, and RDMA data structure work in literature.

Towards that end, these works open the doors to designing approaches that can be added to frameworks or compilers. There is a path towards generalizing routing data structure requests between processors like in Chapter 2, utilizing the `SNAPSHOT` API from Chapter 3 and first-class support for more architectures to design high-performance transactional processing systems, and considering the applicability of heterogeneous architectures to accelerate transaction processing like in Chapter 4. Our frameworks and data structure work in Chap-

ter 5 and 6 has already presented a path forward to extending applicability through CODS.

Concurrent work in compiler design for heterogeneity has made it possible to extend these approaches to become widely accessible. The multi-level intermediate representation project [106] (MLIR) enables straightforward generalization of compiler infrastructure across domains, including CPU and GPU backends. It is a core part of the LLVM project and enables compilation to any architecture that supports LLVM. The MLIR framework has also been applied in the context of data repositories. LingoDB [97], has used this infrastructure for just in time compiling and executing SQL queries. Extending the approaches presented in the prior chapters to general MLIR dialects, when applicable, would allow for the evaluation of ISA affinity on any architecture that supports the LLVM project.

Generalizing these approaches is essential when considering heterogeneous systems. Lowering the burden of exploring ISA affinity can open the doors to further understanding and innovation. This work has begun to open the doors for others to explore these topics through a similar lens within the context of data repositories.

Generalization Across Domains. There are also avenues for building upon the work in this dissertation and applying these techniques to other domains considering similar problems. Similar to what is considered in the work in this dissertation, there are open and emerging problems in machine learning and high-performance systems related to interconnection and synchronization across multiple processors. Specifically, researchers and industry leaders are approaching the issue of machine learning and many similar HPC problems by distributing matrices across processors and performing computation through complex pipelined series of reductions, scatters, and gathers [39, 89, 128]. The design of these systems must consider network topologies, individual GPU kernel-level optimizations, and algorithmic approaches for efficient computation.

Furthermore, there are also interesting new caching designs [196, 148, 90] needed for minimizing the overhead of large language model (LLM) computation. MemServe [90] is designed for disaggregated LLM serving and caches prior computation while focusing on scheduling policies. MemServe utilizes both CPU and GPU memory to cache this computation. Under the hood, it utilizes a radix tree similar to SGLang [196] to cache computation.

The works we presented focus on moving data efficiently between the CPU and GPU

with low latency and high throughput, CPU and GPU synchronization, data structures, and seeking an optimal Pareto frontier for latency and throughput. The problems present in machine learning systems also consider the same issues within a different context. Our contributions to keeping CPU and GPU data structures consistent and designing coalesced data structures are prime candidates for generalization. The strategies presented throughout the prior chapters for managing memory and considering data movement are also generalizable.

Generalizing these approaches across domains by building upon both this body of work and the existing work in machine learning systems can advance the collective understanding of optimal approaches to heterogeneous systems.

7.2 Final Remarks

For the past few years, recognition of the importance of heterogeneity for the future of computation has significantly grown. It will only grow more important as Moore's Law ends. The techniques and methodology presented in this research lay some of the groundwork for how we design data repositories regarding heterogeneity and heterogeneous systems in general.

Bibliography

[1] 4Paradigm, "Openmldb," May 2022. [Online]. Available: https://github.com/4paradigm/OpenMLDB

[2] ——, "Intelligent banking," January 2023. [Online]. Available: https://en.4paradigm.com/industry/banking.html

[3] M. Abebe, H. Lazu, and K. Daudjee, "Proteus: Autonomous adaptive storage for mixed workloads," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 700–714. [Online]. Available: https://doi.org/10.1145/3514221.3517834

[4] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, "Uniswap v3 core," March 2021. [Online]. Available: https://uniswap.org/whitepaper-v3.pdf

[5] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, MIT, 1999.

[6] M. Alam, S. B. Yoginath, and K. S. Perumalla, "Performance of point and range queries for in-memory databases using radix trees on gpus," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2016, pp. 1493–1500.

[7] S. Alam, H. Kamal, and A. Wagner, "A scalable distributed skip list for range queries," in *The 23rd International Symposium on High-Performance Parallel and*

Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014, B. Plale, M. Ripeanu, F. Cappello, and D. Xu, Eds. ACM, 2014, pp. 315–318. [Online]. Available: https://doi.org/10.1145/2600212.2600712

[8] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 577–591. [Online]. Available: https://doi.org/10.1145/2694344.2694391

[9] "Smart contracts," Algorand, Inc., May 2023. [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts

[10] "Transactions," Algorand, Inc., May 2023. [Online]. Available: https://developer.algorand.org/docs/get-details/transactions

[11] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 576–585.

[12] Amazon, May 2021. [Online]. Available: https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html

[13] AMD, May 2021. [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x

[14] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki, "The case for heterogeneous HTAP," in *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf

[15] M. Arbel and H. Attiya, "Concurrent updates with rcu: search tree as an example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 196–205. [Online]. Available: https://doi.org/10.1145/2611462.2611471

[16] M. Arbel-Raviv and T. Brown, "Harnessing epoch-based reclamation for efficient range queries," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 14–27, 2018.

[17] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 320–332. [Online]. Available: https://doi.org/10.1145/3079856.3080231

[18] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 419–429.

[19] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53–64, Jun. 2012. [Online]. Available: https://doi.org/10.1145/2318857.2254766

[20] H. Avni, N. Shavit, and A. Suissa, "Leaplist: lessons learned in designing tm-supported range queries," in *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, P. Fatourou and G. Taubenfeld, Eds. ACM, 2013, pp. 299–308. [Online]. Available: https://doi.org/10.1145/2484239.2484254

[21] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance GPU b-tree," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K.

Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 145–157. [Online]. Available: https://doi.org/10.1145/3293883.3295706

[22] M. A. Awad, S. Ashkiani, S. D. Porumbescu, M. Farach-Colton, and J. D. Owens, "Analyzing and implementing GPU hash tables," in *2023 Symposium on Algorithmic Principles of Computer Systems, APOCS 2023, Florence, Italy, January 25, 2023*, J. Gao, Ed. SIAM, 2023, pp. 33–50. [Online]. Available: https://doi.org/10.1137/1.9781611977578.ch3

[23] M. A. Awad, S. D. Porumbescu, and J. D. Owens, "A gpu multiversion b-tree," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 481–493. [Online]. Available: https://doi.org/10.1145/3559009.3569681

[24] A. Baran, dePaul Miller, E. Lavi, J. Nelson, M. Spear, and R. Palmieri, "Remus," August 2024. [Online]. Available: https://github.com/sss-lehigh/remus

[25] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy, "Kiwi: A key-value map for scalable real-time analytics," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3399718

[26] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. [Online]. Available: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[28] BlazingDB, "blazingsql," 2024. [Online]. Available: https://github.com/BlazingDB/blazingsql

[29] G. E. Blelloch and Y. Wei, "VERLIB: concurrent versioned pointers," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of*

Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024, M. Steuwer, I. A. Lee, and M. Chabbi, Eds. ACM, 2024, pp. 200–214. [Online]. Available: https://doi.org/10.1145/3627535.3638501

[30] R. Bordawekar and P. G. D'souza, "Evaluation of hybrid cache-coherent concurrent hash table on power9 system with nvlink 2," 2018. [Online]. Available: http://on-demand.gputechconf.com/gtc/2018/video/S8172/

[31] A. Brownsword, W. W. Fung, I. Singh, and T. M. Aamodt, "Kilo tm: Hardware transactional memory for gpu architectures," *IEEE Micro*, vol. 32, pp. 7–16, 03 2012. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MM.2012.16

[32] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, p. 777–786, aug 2004. [Online]. Available: https://doi.org/10.1145/1015706.1015800

[33] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," *http://ethereum.org/ethereum.html*, 2014.

[34] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies*, 2020, p. 209.

[35] D. Castro, P. Romano, A. Illic, and A. M. Khan, "Hetm: Transactional memory for heterogeneous systems," 2019.

[36] CCCL Development Team, *CCCL: CUDA C++ Core Libraries*, 2023. [Online]. Available: https://github.com/NVIDIA/cccl

[37] D. Cederman, P. Tsigas, and M. T. Chaudhry, "Towards a software transactional memory for graphics processors." in *EGPGV*, 2010, pp. 121–129.

[38] A. Cerone and A. Gotsman, "Analysing snapshot isolation," *Journal of the ACM (JACM)*, vol. 65, no. 2, pp. 1–41, 2018.

[39] L.-W. Chang, W. Bao, Q. Hou, C. Jiang, N. Zheng, Y. Zhong, X. Zhang, Z. Song, C. Yao, Z. Jiang, H. Lin, X. Jin, and X. Liu, "Flux: Fast software-based communication overlap on gpus through kernel fusion," 2024. [Online]. Available: https://arxiv.org/abs/2406.06858

[40] B. Chatterjee, "Lock-free linearizable 1-dimensional range queries," in *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ser. ICDCN '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3007748.3007771

[41] C. Chen, J. Yang, M. Lu, T. Wang, Z. Zheng, Y. Chen, W. Dai, B. He, W.-F. Wong, G. Wu, Y. Zhao, and A. Rudoff, "Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory," *Proc. VLDB Endow.*, vol. 14, no. 5, pp. 799–812, 2021.

[42] S. Chen, L. Peng, and S. Irving, "Accelerating gpu hardware transactional memory with snapshot isolation," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 282–294.

[43] J. H. Clark, "The geometry engine: A vlsi geometry system for graphics," in *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '82. New York, NY, USA: Association for Computing Machinery, 1982, p. 127–133. [Online]. Available: https://doi.org/10.1145/800064.801272

[44] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas, "The mixed workload ch-benchmark," in *DBTest '11*. ACM, 2011.

[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[47] I. corp, "Intel 64 and ia-32 architectures software developer's manual volume 2: Instruction set reference," 2021.

[48] T. P. P. Council, "Tpc benchmark c revision 5.11," February 2010.

[49] ——, "Tpc benchmark h revision 3.0.1," April 2022.

[50] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS '13. USA: IEEE Computer Society, 2013, p. 196–205. [Online]. Available: https://doi.org/10.1109/ICDCS.2013.42

[51] H. Daly, A. Hassan, M. F. Spear, and R. Palmieri, "NUMASK: High Performance Scalable Skip List for NUMA," in *32nd International Symposium on Distributed Computing (DISC 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), U. Schmid and J. Widder, Eds., vol. 121. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 18:1–18:19. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9807

[52] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[53] dePaul Miller, J. Nelson, A. Hassan, and R. Palmieri, "KVCG: a heterogeneous key-value store for skewed workloads," in *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, B. Wassermann, M. Malka, V. Chidambaram, and D. Raz, Eds. ACM, 2021, pp. 5:1–5:12. [Online]. Available: https://doi.org/10.1145/3456727.3463779

[54] T. Derei, "Accelerating the plonk zksnark proving system using gpu architectures," Master's thesis, Lehigh University, 2023.

[55] T. Derei, B. Aulenbach, V. Carolino, C. Geren, M. Kaufman, J. Klein, R. Islam Shanto, and H. F. Korth, "Scaling zero-knowledge to verifiable databases," in *Proceedings of the 1st Workshop on Verifiable Database Systems*, ser. VDBS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–9. [Online]. Available: https://doi.org/10.1145/3595647.3595648

[56] I. Dick, A. Fekete, and V. Gramoli, "A skip list for multicore," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, p. e3876, 2017.

[57] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, oct 2021. [Online]. Available: https://doi.org/10.1145/3483840

[58] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds. USENIX Association, 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi\%C4\%87

[59] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[60] L. Eeckhout, "Is moore's law slowing down? what's next?" *IEEE Micro*, vol. 37, no. 04, pp. 4–5, 2017.

[61] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications,*

technologies, architectures, and protocols for computer communication, 2012, pp. 25–36.

[62] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, p. 365–376, jun 2011. [Online]. Available: https://doi.org/10.1145/2024723.2000108

[63] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976. [Online]. Available: https://doi.org/10.1145/360363.360369

[64] "Proof-of-work mining with ethash: Go," Ethereum Foundation, June 2023. [Online]. Available: https://geth.ethereum.org/docs/fundamentals/mining

[65] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[66] A. Foundation, "Aptos core," Dec 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core

[67] ——, "Aptos developer documentation," Jul 2023. [Online]. Available: https://aptos.dev/

[68] S. Franey and M. H. Lipasti, "Accelerating atomic operations on gpgpus," in *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS), Tempe, AZ, USA, April 21-24, 2013*. IEEE, 2013, pp. 1–8. [Online]. Available: https://doi.org/10.1109/NoCS.2013.6558404

[69] W. W. Fung and T. M. Aamodt, "Energy efficient gpu transactional memory via space-time optimizations," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 408–420.

[70] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 2, pp. 186–213, 1983.

[71] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy, Eds. ACM, 2023, pp. 232–244. [Online]. Available: https://doi.org/10.1145/3572848.3577524

[72] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovici, and R. Shahout, "Evendb: optimizing key-value storage for spatial locality," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 27:1–27:16. [Online]. Available: https://doi.org/10.1145/3342195.3387523

[73] Google, "Leveldb," https://github.com/google/leveldb, 2019.

[74] Google, May 2021. [Online]. Available: https://cloud.google.com/gpu

[75] Google, "Introducing alloydb for postgresql: Free yourself from expensive, legacy databases," May 2022. [Online]. Available: https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql

[76] ——, "Leveldb," 2024. [Online]. Available: https://github.com/google/leveldb

[77] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688501

[78] T. P. G. D. Group, "Transaction isolation," November 2022. [Online]. Available: https://www.postgresql.org/docs/current/transaction-iso.html

[79] Y. Guo and G. Li, "A CXL- powered database system: Opportunities and challenges," in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 5593–5604. [Online]. Available: https://doi.org/10.1109/ICDE60146.2024.00447

[80] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent gpu-accelerated DNN inferences," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 539–558. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/han

[81] M. Harris, "Unified memory in cuda 6," November 2013. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[82] ——, "Gpu pro tip: Cuda 7 streams simplify concurrency," January 2015. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[83] ——, "How to access global memory efficiently in cuda c/c kernels," May 2018. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

[84] M. Harris and K. Perelygin, "Cooperative groups: Flexible cuda thread programming," Oct 2017. [Online]. Available: https://developer.nvidia.com/blog/cooperative-groups/

[85] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Welch, Ed., vol. 2180. Springer, 2001, pp. 300–314. [Online]. Available: https://doi.org/10.1007/3-540-45414-4_21

[86] HEAVY.AI, "Heavydb," 2024. [Online]. Available: https://github.com/heavyai/heavydb

[87] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. H. Anderson, G. Prencipe, and R. Wattenhofer, Eds., vol. 3974. Springer, 2005, pp. 3–16. [Online]. Available: https://doi.org/10.1007/11795490_3

[88] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

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

[90] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun, and Y. Shan, "Memserve: Context caching for disaggregated llm serving with elastic memory pool," 2024. [Online]. Available: https://arxiv.org/abs/2406.17565

[91] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "Tidb: A raft-based htap database," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3072–3084, 2020.

[92] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database," *IEEE Data Engineering Bulletin*, 2012.

[93] "Intel® threading building blocks developer guide," Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-guide.html

[94] "Intel core i5-10600 processor," Intel, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/199273/intel-core-i510600-processor-12m-cache-up-to-4-80-ghz/specifications.html

[95] D. Interactive, "Memcached," October 2022. [Online]. Available: https://github.com/memcached/memcached

[96] L. Jenkins, T. Zhou, and M. Spear, "Redesigning go's built-in map to support concurrent operations," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 14–26.

[97] M. Jungmair, A. Kohn, and J. Giceva, "Designing an open framework for query optimization and compilation," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2389–2401, Jul. 2022. [Online]. Available: https://doi.org/10.14778/3551793.3551801

[98] A. Kalia, M. Kaminsky, and D. G. Andersen, "{FaSST}: Fast, scalable and simple distributed transactions with {Two-Sided}({{{{{RDMA}}}}}) datagram {RPCs}," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.

[99] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. K. Ousterhout, "SLIK: scalable low-latency indexes for a key-value store," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kejriwal

[100] A. Kerr, H. Wu, M. Gupta, D. Blasig, P. Ramini, D. Merrill, A. Shivam, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Nicely, "CUTLASS." [Online]. Available: https://github.com/NVIDIA/cutlass

[101] J. Kim, J. Yu, J. Ahn, S. Kang, and H. Jung, "Diva: Making mvcc systems htap-friendly," in *SIGMOD '22*. ACM, 2022, p. 49–64.

[102] M. J. Kishi, S. Peluso, H. F. Korth, and R. Palmieri, "SSS: scalable key-value store with external consistent and abort-free read-only transactions," in *39th*

IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019. IEEE, 2019, pp. 589–600. [Online]. Available: https://doi.org/10.1109/ICDCS.2019.00065

[103] H. F. Korth, E. Levy, and A. Silberschatz, "A formal approach to recovery by compensating transactions," in *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, D. McLeod, R. Sacks-Davis, and H. Schek, Eds. Morgan Kaufmann, 1990, pp. 95–106. [Online]. Available: http://www.vldb.org/conf/1990/P095.PDF

[104] A. Krishna Srinivasan and P. D. Director, "Opening remarks at peer-learning series on digital money/technology: Central bank digital currency and the case of china," Jul 2022. [Online]. Available: https://www.imf.org/en/News/Articles/2022/07/07/sp070722-central-bank-digital-currency-and-the-case-of-china#.ZD2fh81KeHA.link

[105] A. Kumar and M. Stonebraker, "Semantics based transaction management techniques for replicated data," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 117–125, 1988.

[106] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[107] R. Lee, M. Zhou, C. Li, S. Hu, J. Teng, D. Li, and X. Zhang, "The art of balance: A rateupdb experience of building a CPU/GPU hybrid database product," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 2999–3013, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p2999-lee.pdf

[108] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 109–118.

[109] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 1092–1098, 2015.

[110] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "{MICA}: A holistic approach to fast in-memory key-value storage," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 429–444.

[111] G. H. Loh, S. Naffziger, and K. Lepak, "Understanding chiplets today to anticipate future integration opportunities and limits," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 142–145.

[112] G. H. Loh and Y. Xie, "3d stacked microprocessor: Are we there yet?" *IEEE Micro*, vol. 30, no. 3, pp. 60–64, 2010.

[113] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–270. [Online]. Available: https://doi.org/10.1145/3297858.3304043

[114] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu, "GZKP: A GPU accelerated zero-knowledge proof system," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 340–353. [Online]. Available: https://doi.org/10.1145/3575693.3575711

[115] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," *Stellar Development Foundation*, vol. 32, pp. 1–45, 2015.

[116] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, "A transaction model for multidatabase systems," in *ICDCS*, 1992, pp. 56–63.

[117] H. Meir, D. Basin, E. Bortnikov, A. Braginsky, Y. Gottesman, I. Keidar, E. Meir, G. Sheffi, and Y. Zuriel, "Oak: a scalable off-heap allocated key-value map," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, R. Gupta and X. Shen, Eds. ACM, 2020, pp. 17–31. [Online]. Available: https://doi.org/10.1145/3332466.3374526

[118] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: ACM, 2002, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/564870.564881

[119] Microsoft, "Microsoft data platform." [Online]. Available: https://www.microsoft.com/en-us/sql-server

[120] "Azure vm sizes - gpu - azure virtual machines," Microsoft, May 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu

[121] C. Mims, "Huang's law is the new Moore's law, and explains why Nvidia wants Arm," *The Wall Street Journal*, Sep 2020. [Online]. Available: https://www.wsj.com/articles/huangs-law-is-the-new-moores-law-and-explains-why-nvidia-wants-arm-11600488001

[122] P. Misra and M. Chaudhuri, "Performance evaluation of concurrent lock-free data structures on gpus," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 53–60.

[123] G. Mitenkov, I. Kabiljo, Z. Li, A. Spiegelman, S. Vusirikala, Z. Xiang, A. Zlateski, N. P. Lopes, and R. Gelashvili, "Deferred objects to enhance smart contract programming with optimistic parallel execution," 2024.

[124] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[125] N. Moscovici, N. Cohen, and E. Petrank, "A gpu-friendly skiplist algorithm," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Ieee, 2017, pp. 246–259.

[126] MySQL, May 2021. [Online]. Available: https://www.mysql.com/

[127] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

[128] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476209

[129] J. Nelson, dePaul Miller, and R. Palmieri, "Don't forget about synchronization! guidelines for using locks on graphics processing units," *Concurr. Comput. Pract. Exp.*, vol. 34, no. 2, 2022. [Online]. Available: https://doi.org/10.1002/cpe.5757

[130] J. Nelson-Slivon, A. Hassan, and R. Palmieri, "Bundling linked data structures for linearizable range queries," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 368–384. [Online]. Available: https://doi.org/10.1145/3503221.3508412

[131] J. Nelson-Slivon, R. Yankovich, A. Hassan, and R. Palmieri, "Brief announcement: Rome: Wait-free objects for rdma," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY,

USA: Association for Computing Machinery, 2024, p. 371–373. [Online]. Available: https://doi.org/10.1145/3626183.3660262

[132] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 677–689. [Online]. Available: https://doi.org/10.1145/2723372.2749436

[133] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[134] "Cuda toolkit documentation," NVIDIA, 2018. [Online]. Available: https://docs.nvidia.com/cuda/archive/9.1/

[135] NVIDIA, May 2021. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/

[136] "Cuda c programming guide," NVIDIA, May 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[137] NVIDIA, "Cuda c++ best practices guide," March 2022. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.6.2/cuda-c-best-practices-guide/index.html

[138] ——, "Ptx isa 8.3," Nov 2023. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution

[139] ——, Sep 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide

[140] NVIDIA, "Connectx-8 supernic," August 2024. [Online]. Available: https://nvdam.widen.net/s/pxsjzhgw6j/connectx-datasheet-connectx-8-supernic-3231505

[141] NVIDIA, *cuCollections*, 2024. [Online]. Available: https://github.com/NVIDIA/cuCollections

[142] ——, "Cuda zone," Jan 2024. [Online]. Available: https://developer.nvidia.com/cuda-zone

[143] NVIDIA, "Nvidia h200 tensor core gpu," Aug 2024. [Online]. Available: https://nvdam.widen.net/s/nb5zzzsjdf/hpc-datasheet-sc23-h200-datasheet-3002446

[144] ——, "Nvidia hierarchicalkv," August 2024. [Online]. Available: https://github.com/NVIDIA-Merlin/HierarchicalKV

[145] Oracle, "Oracle database 19c," January 2023. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/index.html

[146] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann, "Fast scans on key-value stores," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1526–1537, aug 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137659

[147] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1493–1508. [Online]. Available: https://doi-org.ezproxy.lib.lehigh.edu/10.1145/2723372.2747645

[148] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," in *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen, Eds., vol. 5. Curan, 2023, pp. 606–624. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf

[149] G. Prasaad, A. Cheung, and D. Suciu, "Handling highly contended OLTP workloads using fast dynamic partitioning," in *Proceedings of the 2020 International Conference*

on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 527–542. [Online]. Available: https://doi.org/10.1145/3318464.3389764

[150] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[151] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 246–257.

[152] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 65–76.

[153] G. Ramseyer, A. Goel, and D. Mazières, "Speedex: A scalable, parallelizable, and economically efficient decentralized exchange," 2023.

[154] G. Ramseyer and D. Mazières, "Groundhog: Linearly-scalable smart contracting via commutative transaction semantics," *arXiv preprint arXiv:2404.03201*, 2024.

[155] Redis, "Redis," 2023. [Online]. Available: https://redis.io/docs/

[156] M. Rodriguez, A. Hassan, and M. Spear, "Exploiting locality in scalable ordered maps," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 351–352. [Online]. Available: https://doi.org/10.1145/3410463.3414665

[157] K. A. Ross, "Efficient hash probes on modern processors," in *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, R. Chirkova, A. Dogac, M. T. Özsu, and T. K.

Sellis, Eds. IEEE Computer Society, 2007, pp. 1297–1301. [Online]. Available: https://doi.org/10.1109/ICDE.2007.368997

[158] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri, "Processing transactions in a predefined order," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 120–132. [Online]. Available: https://doi.org/10.1145/3293883.3295730

[159] M. M. Saad, R. Palmieri, A. Hassan, and B. Ravindran, "Extending tm primitives using low level semantics," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–120. [Online]. Available: https://doi.org/10.1145/2935764.2935794

[160] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. VLDB Endow.*, vol. 11, no. 4, p. 420–431, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3186728.3164139

[161] N. Sakharnykh, "Everything you need to know about unified memory," 2018. [Online]. Available: http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[162] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1617–1632. [Online]. Available: https://doi.org/10.1145/3318464.3380595

[163] D. Shankar, X. Lu, and D. K. D. Panda, "Simdht-bench: Characterizing simd-aware hash table designs on emerging cpu architectures," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 178–188.

[164] S. Shen, R. Chen, H. Chen, and B. Zang, "Retrofitting high availability mechanism to tame hybrid transaction/analytical processing," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 219–238. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/shen

[165] ——, "Vegito: a fast distributed in-memory htap system," Dec 2021. [Online]. Available: https://github.com/SJTU-IPADS/vegito

[166] D. Siakavaras, P. Billis, K. Nikas, G. I. Goumas, and N. Koziris, "Efficient concurrent range queries in b+-trees using RCU-HTM," in *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, C. Scheideler and M. Spear, Eds. ACM, 2020, pp. 571–573. [Online]. Available: https://doi.org/10.1145/3350755.3400237

[167] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. [Online]. Available: https://www.db-book.com/

[168] SingleStore, "Singlestore." [Online]. Available: https://www.singlestore.com

[169] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. J. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia, "DBOS: A dbms-oriented operating system," *Proc. VLDB Endow.*, vol. 15, no. 1, pp. 21–30, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p21-skiadopoulos.pdf

[170] "Operations and transactions," Stellar Development Foundation, Jan 2023. [Online]. Available: https://developers.stellar.org/docs/fundamentals-and-concepts/stellar-data-structures/operations-and-transactions

[171] "Soroban," Stellar Development Foundation, Jun 2023. [Online]. Available: https://soroban.stellar.org/docs

[172] M. Stonebraker and L. A. Rowe, "The design of postgres," *ACM Sigmod Record*, vol. 15, no. 2, pp. 340–355, 1986.

[173] R. D. Team, *RAPIDS: Libraries for End to End GPU Data Science*, 2023. [Online]. Available: https://rapids.ai

[174] J. Thomas, "Nvidia geforce gtx 1660 super review," April 2022. [Online]. Available: https://www.techradar.com/reviews/nvidia-geforce-gtx-1660-super

[175] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 1–12.

[176] B. Tong, Y. Zhou, C. Zhang, J. Tang, J. Tang, L. Yang, Q. Li, M. Lin, Z. Bao, J. Li, and L. Chen, "Galaxybase: A high performance native distributed graph database for HTAP," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 3893–3905, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p3893-tong.pdf

[177] Uber, "Aresdb," 2024. [Online]. Available: https://github.com/uber/aresdb

[178] "Federal reserve board releases discussion paper that examines pros and cons of a potential u.s. central bank digital currency (cbdc)," United States of America Federal Reserve, Jan 2022. [Online]. Available: https://www.federalreserve.gov/newsevents/pressreleases/other20220120a.htm

[179] M. Vezenov, "Accelerating zksnarks on modern architectures," Master's thesis, Lehigh University, 2022.

[180] H. Wang, H. Dai, S. Chen, and G. Chen, "Rethinking hash tables: Challenges and opportunities with compute express link (CXL)," in *ACM Turing Award Celebration Conference 2024, ACM-TURC 2024, Changsha, China, July 5-7, 2024*. ACM, 2024. [Online]. Available: https://doi.org/10.1145/3674399.3674418

[181] X. Wei, R. Chen, H. Chen, and B. Zang, "Xstore: Fast rdma-based ordered key-value store using remote learned cache," *ACM Trans. Storage*, vol. 17, no. 3, pp. 18:1–18:32, 2021. [Online]. Available: https://doi.org/10.1145/3468520

[182] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing rdma-enabled distributed transactions: Hybrid is better!" in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 233–251. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/wei

[183] X. Wei, H. Wang, T. Wang, R. Chen, J. Gu, P. Zuo, and H. Chen, "Transactional indexes on (RDMA or cxl-based) disaggregated memory with repairable transaction," *CoRR*, vol. abs/2308.02501, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.02501

[184] Y. Wei, N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, and Y. Sun, "Constant-time snapshots with applications to concurrent data structures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–46. [Online]. Available: https://doi.org/10.1145/3437801.3441602

[185] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. [Online]. Available: https://gavwood.com/paper.pdf

[186] S. Xiao and W. Feng, "Inter-block gpu communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[187] Y. Xu, R. Wang, N. Goswami, T. Li, and D. Qian, "Software transactional memory for gpu architectures," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 49–52, Jan 2014.

[188] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 18, no. 2, pp. 41–49, 2013.

[189] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based synchronization for gpu architectures," in *Proceedings of the ACM International Conference on*

Computing Frontiers, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 205–213. [Online]. Available: http://doi.acm.org/10.1145/2903150.2903155

[190] J. Yang, I. Rae, J. Xu, J. Shute, Z. Yuan, K. Lau, Q. Zeng, X. Zhao, J. Ma, Z. Chen, Y. Gao, Q. Dong, J. Zhou, J. Wood, G. Graefe, J. F. Naughton, and J. Cieslewicz, "F1 lightning: HTAP as a service," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3313–3325, 2020.

[191] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 191–208. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/yang

[192] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *PVLDB*, vol. 8, no. 11, pp. 1226–1237, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1226-zhang.pdf

[193] ——, "Megakv github," 2021. [Online]. Available: https://github.com/pzrq/megakv

[194] S. Zhang, J. He, B. He, and M. Lu, "Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures," *Proc. VLDB Endow.*, vol. 6, no. 12, p. 1374–1377, aug 2013. [Online]. Available: https://doi.org/10.14778/2536274.2536319

[195] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 194–209. [Online]. Available: https://doi.org/10.1145/3447786.3456237

[196] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient

execution of structured language model programs," 2024. [Online]. Available: https://arxiv.org/abs/2312.07104

[197] W. Zhong, C. Chen, X. Wu, and S. Jiang, "{REMIX}: Efficient range query for {LSM-trees}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 51–64.

Biography

dePaul Miller was born in New Jersey in May 1998. He went to high school at Boston College High School. He began his undergraduate degrees at Lehigh University in 2016 and graduated Suma Cum Laude during the pandemic in 2020. He holds a B.S. in Computer Science and a B.A. in Mathematics from Lehigh University. He remained at Lehigh University, earning his M.S. in Computer Science in May 2023. During his graduate studies, he interned multiple times at NVIDIA. He also received a fellowship from the Koerner Family Foundation. He is currently employed at NVIDIA as a performance software engineer on the Fast Kernels team, working on the performance of NVIDIA's deep learning libraries. His first-author publications include accepted papers at the ACM International Systems and Storage Conference (SYSTOR), the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), and the ACM SIGPLAN Anual Symposium on Principles and Practice of Parallel Programming (PPoPP).