SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

August 2026, Vol. 69, Iss. 8, 182101:1–182101:18
https://doi.org/10.1007/s11432-025-4735-6

GPUSync: a benchmark suite for fine-grained synchronization on GPUs

Lan GAO[1], Min YUAN[1], Yu GUO[1], Rui WANG[2*], Weigong ZHANG[1] & Depei QIAN[2]

[1]*College of Information Engineering, Capital Normal University, Beijing 100048, China*
[2]*School of Computer Science and Engineering, Beihang University, Beijing 100191, China*

Received 10 April 2025/Revised 28 October 2025/Accepted 17 December 2025/Published online 15 July 2026

Abstract As more emerging applications are moving to GPUs, fine-grained synchronization has become imperative. However, achieving efficient fine-grained synchronization on GPUs remains challenging, driving significant research interest. Despite numerous approaches in research literature, there is still a lack of standardized workloads for comprehensive evaluation and a systematic understanding of how GPU architectures can efficiently support synchronization. Furthermore, two key issues remain unresolved. First, the diverse synchronization behaviors introduce numerous potential scenarios. This presents a significant challenge for comprehensive evaluation. Second, the unique characteristics of GPUs introduce distinct performance-impacting factors. However, there is insufficient insight into how these factors affect the execution efficiency of fine-grained synchronization. In this article, we introduce GPUSync, a benchmark suite designed to advance fine-grained synchronization on GPUs. GPUSync provides eight applications across various domains, each optimized with locking and transactional memory (TM). To facilitate the evaluation, we provide a novel classification method to organize synchronization behaviors and comprehensive evaluation parameters to cover a wide range of scenarios. Utilizing GPUSync, we conduct a performance analysis on two GPU platforms, revealing four key insights into architectural support and software development for fine-grained synchronization, offering valuable guidance for more efficient fine-grained synchronization for GPUs.

Keywords fine-grained synchronization, GPU, benchmark suite, locking, transactional memory

Citation Gao L, Yuan M, Guo Y, et al. GPUSync: a benchmark suite for fine-grained synchronization on GPUs. Sci China Inf Sci, 2026, 69(8): 182101, https://doi.org/10.1007/s11432-025-4735-6

1 Introduction

Synchronization is critical for parallel programming with shared memory. Recently, various parallel applications with data sharing have been moved to run on GPUs due to the massive multi-threading capability and cost-effectiveness. These applications share data among individual threads/warps/TBs (thread blocks) frequently. To fully exploit the massive parallelism, they commonly employ fine-grained synchronization on GPUs, where a small amount of shared data is associated with each mutex (for example, each shared data is protected by a single mutex), such as online transaction processing (OLTP) workloads, database systems, and large graphs and trees that play fundamental roles in many important applications.

Unlike coarse-grained synchronization, fine-grained synchronization on GPUs involves more synchronization operations, making it particularly challenging due to GPUs' architecture characteristics. Firstly, the SIMT (single instruction multiple threads) execution model can easily lead to deadlocks and livelocks, which persist even in modern GPUs with independent thread scheduling [1]. Moreover, synchronization efficiency suffers from two key limitations: (1) the commonly employed atomic primitives are inefficient due to the GPU's relaxed memory model [2], and (2) frequent branch divergences and busy-waiting mechanisms introduce substantial performance overhead [3, 4]. These challenges have motivated extensive research on both correctness [1, 5–14] and performance optimization [2–4, 15] of fine-grained synchronization on GPUs. For instance, notable efforts include NVIDIA's hardware modifications in Volta GPUs to support fine-grained locking [16]. Despite these advances, current GPU architectures still lack robust fine-grained synchronization mechanisms, with neither hardware nor programming models providing efficient primitives for shared data access.

Meanwhile, existing evaluations of fine-grained synchronization on GPU architectures predominantly rely on either microbenchmarks or applications from benchmark suites like NVIDIA SDK [17] and Rodinia [18]. While

* Corresponding author (email: wangrui@buaa.edu.cn)

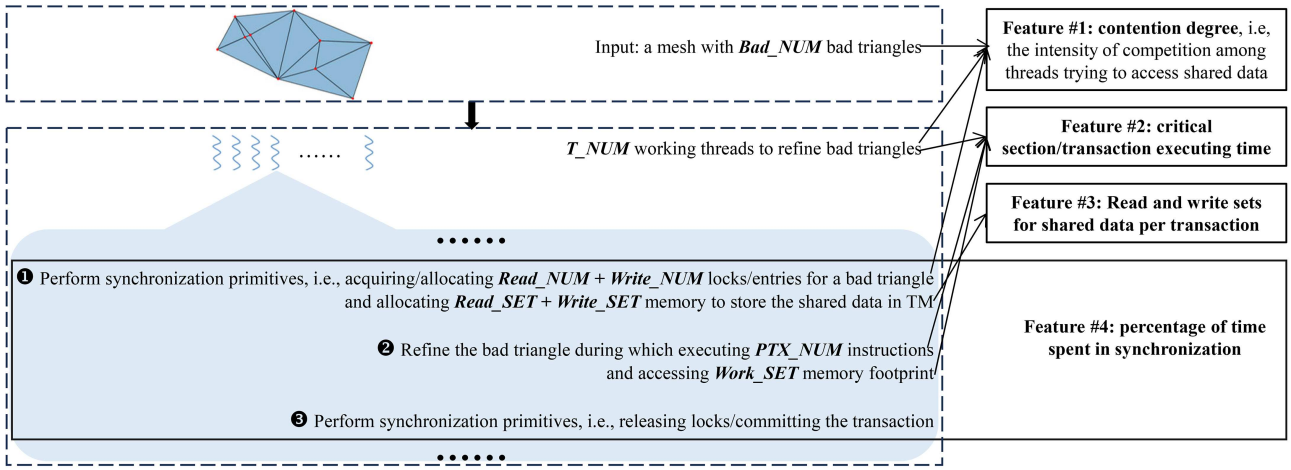

Figure 1 (Color online) A typical example of a fine-grained synchronization application (Delaunay mesh refinement, DMR) on GPUs, to illustrate the diverse synchronization behaviors (bold italic text) and our four key features (bold text).

microbenchmarks are useful in targeting specific system features, they are not representative of how realistic applications behave. On the other hand, the applications in these benchmarks are heavily optimized to minimize synchronization and communication across threads. Consequently, these applications fail to capture essential synchronization properties on GPUs.

Moreover, there remain two critical unresolved issues in the evaluation of fine-grained synchronization on GPUs. Firstly, as shown in Figure 1, applications with fine-grained synchronization have diverse synchronization behaviors, where modifications to individual components or their interactions may yield distinct synchronization characteristics. This creates an exponentially large configuration space, and it becomes particularly complex on GPUs as many synchronization behaviors span a vast parameter space. For example, the number of parallel threads that compete for shared data can reach tens of thousands for applications with fine-grained synchronization on GPUs. Such scale and complexity necessitate systematic behavior characterization to (1) ensure comprehensive coverage of diverse synchronization scenarios for a thorough analysis of fine-grained synchronization properties on GPUs, and (2) avoid redundant evaluation of similar synchronization characteristics. Unfortunately, existing studies lack this systematic characterization.

Secondly, to support the simultaneous execution of numerous threads, GPUs employ a hierarchical hardware organization and SIMT execution paradigm, and provide abundant fast on-chip memories and high memory bandwidth. However, there is no detailed empirical analysis of how these unique characteristics impact the fine-grained synchronization efficiency. Only with this type of empirical analysis can one identify the bottlenecks and get insights into how to design better fine-grained synchronization mechanisms for GPU architectures.

To this end, we develop GPUSync, a fine-grained synchronization benchmark suite designed for GPUs. By enabling systematic exploration of GPU-specific synchronization architectural support and offering a standardized platform for evaluating fine-grained synchronization, GPUSync aims to advance the research of fine-grained synchronization for GPU architectures. In GPUSync, we integrate eight carefully selected applications in a wide range of algorithms and application domains (part ③ in Figure 2). These applications all show natural massive parallelism with a lot of shared data being processed. To exhibit different synchronization characteristics, we modify each of them with specialized data structures, tailored parallelization strategies, and efficient synchronization schemes (illustrated in detail in Subsection 4.1). Taking the widely used scientific application DMR as an example, it is well-suited for GPU computing due to its massive number of triangles requiring refinement. Unlike the traditional DMR algorithm on GPUs [19], which pre-selects conflicting triangles before refinement to avoid synchronization, we assign each thread to process a triangle independently, and ensure the correctness via fine-grained synchronization to show its synchronization characteristics. In addition, we implement each application with two fundamental synchronization techniques: locking and TM mechanisms, and support all the fine-grained synchronization schemes implemented with the standard APIs (part ④ Figure 2).

To facilitate the evaluation, we categorize the diverse synchronization behaviors into four key features (as shown in Figure 1), and describe each of them quantitatively or qualitatively through dedicated metrics. Based on this categorization, we then design comprehensive evaluation parameters for all the applications in GPUSync to ensure broad coverage of synchronization scenarios. We also present a detailed synchronization characterization for the GPUSync applications (part ② in Figure 2). Specifically, we quantify the contention degree using the ideal minimal

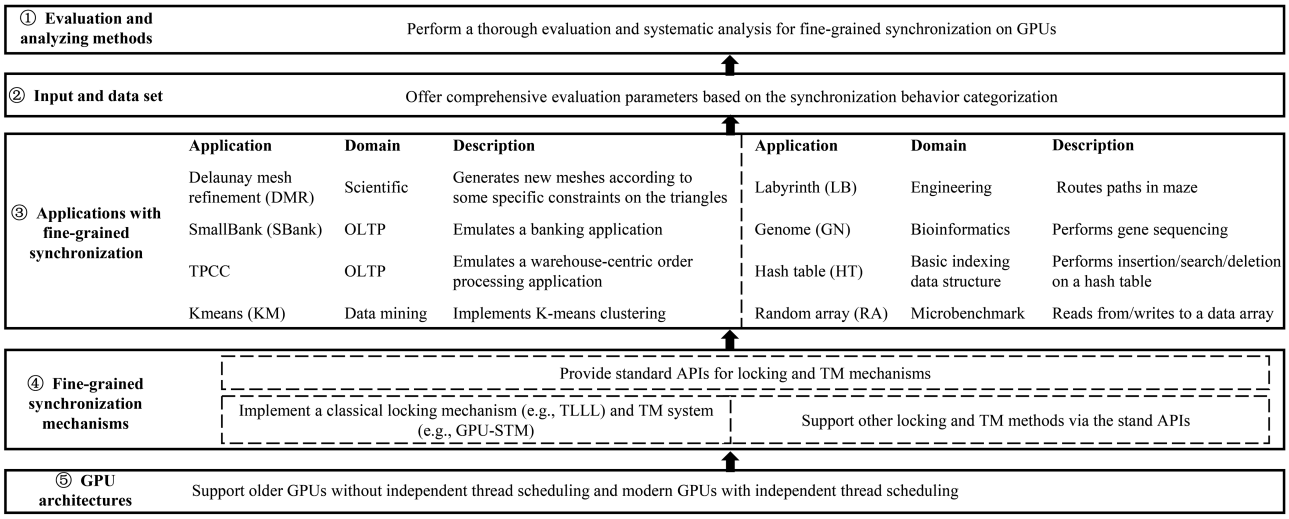

Figure 2 GPUSync framework.

synchronization failure ratio (i.e., the transaction abort rate in TM systems and the lock contention rate in lock-based implementations), and qualitatively describe the critical section/transaction[1)] executing time with critical section/transaction length, measure the sizes of read and write sets for TM, and evaluate the percentage of time spent in fine-grained synchronization.

Furthermore, to demonstrate the usefulness of GPUSync for GPU fine-grained synchronization research, we run it using both a classical locking and TM system (part ④ in Figure 2). We also adapt these schemes to run successfully on both old and modern GPUs (part ⑤ in Figure 2). Additionally, we present a thorough evaluation methodology by carefully filtering out the metrics necessary for assessing the performance of applications with fine-grained synchronization. Specifically, we evaluate the performance and synchronization failure ratio, explore the performance influencing factors, and discuss the technical issues behind unconventional results (part ① in Figure 2).

Using GPUSync, we find four pivotal insights for fine-grained synchronization on GPUs: (1) independent thread scheduling can introduce indeterminate errors for fine-grained synchronization that rely on the SIMT execution paradigm, and lead to poorer performance for some synchronization scenarios; (2) while GPUs' massive parallelism enables high throughput, it simultaneously amplifies synchronization challenges; (3) fine-grained synchronization performance is particularly sensitive to memory access latency; and (4) branch divergence significantly degrades performance of fine-grained synchronization. These insights are valuable for both the architectural support and software development for fine-grained synchronization on GPUs. GPUSync is publicly available at https://github.com/tlll-gpu/GPUSync.

2 Background

2.1 GPU architecture

In this subsection, we introduce the GPU characteristics that are related to the correctness and performance of fine-grained synchronization applications. NVIDIA GPUs are used without losing generality.

The SIMT execution paradigm of GPUs. GPUs of Pascal and earlier microarchitectures employ the SIMT execution paradigm, where multiple different threads concurrently execute the same set of instructions but with different data. In these GPUs, threads in divergent paths cannot communicate or exchange data with each other, leading to deadlocks (a situation where two threads are mutually blocked, each holding a lock that the other needs to proceed). To ensure correctness, various deadlock detection and avoidance methods have been proposed on these GPUs [1,5].

To improve the programmability for fine-grained synchronization algorithms on GPUs, NVIDIA introduces independent thread scheduling in Volta microarchitecture, and inherits this enhancement in the later microarchitectures (i.e., Turing, Ampere, and Hopper). It can prevent partial deadlocks for fine-grained synchronization, yet it may

1) Both refer to a sequence of synchronization issues where shared data must be accessed atomically by concurrent threads. The critical section is utilized within locking mechanisms, whereas the transaction is specific to TM systems.

```
1: done = false;
2: while (!done){
3:     //acquire 2 locks before executing the transaction
4:     if (lock(mutex0)){
5:         if (lock(mutex1)){
6:             //execute the critical section
7:             ……
8:             done = true;
9:             /* release obtained locks after 1. executing the critical section,
10:                or 2. a lock is failed to be acquired */
11:            release_lock(mutex0, mutex1);
12:        } else {release_lock(mutex0);}
13:    }
14:}
```

(a)

```
1:  done = false;
2:  while (!done){
3:      //initialize the transaction
4:      TXBegin();
5:      TXRead(&shared_data_a);
6:      //to support transaction aborts
7:      if (isOpaque[threadIdxInWarp])   break;
8:      TXWrite(&shared_data_b,  val);
9:      //execute other parts of the transaction
10:     ……
11:     //commit the transaction
12:     done = TXCommit(warp);
13: }
```

(b)

Figure 3 (a) An example of locking; (b) an example of TM.

introduce indeterminate errors for algorithms that rely on the SIMT execution paradigm. To be noticed, while CUDA offers several primitives to facilitate the warp-level convergence, effectively using them to prevent these errors is more complex than initially thought (discussed in detail in Subsection 5.1). Additionally, it is worthwhile to explore how independent thread scheduling can impact the performance of fine-grained synchronization on GPUs.

The branch divergence of GPUs. Due to the SIMT execution paradigm, the execution of different control flow paths will be sequentialized when a branch instruction is executed in a warp, so-called branch divergence. It leads to poor performance and low hardware utilization on GPUs, including those with independent thread scheduling [16]. In GPU applications that perform fine-grained synchronization among individual threads, synchronization failures of some threads within warps can frequently result in branch divergences. Identifying the impact of these divergences on the execution efficiency of fine-grained synchronization is crucial.

The memory system of GPUs. GPUs achieve high throughput by executing a large number of threads concurrently. To facilitate this, GPUs provide abundant fast on-chip memories, such as registers, L1 and L2 caches, and shared memory. Among all the on-chip memories, usage for the register (for fast context switching across threads) of each thread and the shared memory (for fast communication between threads in the same TB) in TBs are two critical factors to determine the number of threads that can run concurrently on a streaming multiprocessor (SM). The lower number of parallel threads that an SM can support results in decreased memory latency hiding, and thereby impacts performance. For applications with fine-grained synchronization, the synchronization operation may lower this number by needing more extra registers and shared memory.

2.2 Fine-grained synchronization on GPUs

Synchronization is required when data is shared among concurrent threads. To exploit the massive multi-thread capability of GPUs, fine-grained synchronization is employed by GPU applications, in which a small amount of shared data is associated with each mutex. The two most common synchronization techniques are locking and TM. As shown in Figure 3(a), in the locking mechanism, the lock needs to be acquired before entering a critical section to guarantee the correctness, and released after the critical section to make the shared data accessible. With the fine-grained synchronization, more than one lock is usually needed for critical sections, so-called nested locks. Figure 3(b) shows how the TM system realizes the synchronization. It encapsulates synchronization issues with transactions by (1) replacing locking with atomic execution units and (2) providing a mechanism to group together all the accesses to shared data in the transaction with the effect that either all of them appear to execute or none of them does. Considering locking and TM are the two most common synchronization techniques, we developed the GPUSync benchmark suite with both of them.

GPUs own thousands of hardware threads and employ the SIMT execution paradigm, making it challenging to perform fine-grained synchronization. Despite extensive research efforts, significant challenges still exist. There is an ongoing need for standardized workloads for comprehensive evaluation, and systematic analysis to derive insights on optimizing fine-grained synchronization for GPU architectures.

3 Related work

There have been many efforts to create benchmarks on GPUs for architectural evaluation with different emphases. As shown in Table 1, SDK [17] was developed by NVIDIA and intended to show the applicability of GPUs; Rodinia [18]

Table 1 Summary of publicly available benchmark suites for GPU architectures.

GPU benchmarks	Target	Applications with synchronization
SDK [17]	GPU applicability	A little
Rodinia [18]	Heterogeneous computing	A little
Parboil [20]	Throughput computing	No
LonestarGPU [21]	Irregular computing	No
PolyBench [22]	Heterogeneous analytics systems	No
SHOC [23]	Performance and stability test for scalable heterogeneous computing	No
Cactus [24]	Complex real-life applications with many kernels	No
Chai [25]	Collaborative heterogeneous applications for integrated architectures	No
Hetero-Mark [26]	CPU-GPU collaborative computing	No
DeepBench [27]	Deep neural network training	No
MLPerf [28]	Machine learning	No
Tango [29]	Deep neural networks	No
TBD [30]	Deep neural network training	No
Altis [31]	Modern GPGPU computing	No
HeteroSync [32]	Micro benchmarks with fine-grained synchronization on integrated GPUs	All
GPUSync	Various applications with fine-grained synchronization on district GPUs	All

was presented for exploring the heterogeneous computing infrastructures which does not include applications with extensive synchronization; Parboil [20] was to study the performance of throughput computing architecture and compilers; LonestarGPU [21], PolyBench [22], Pannotia [33], SHOC [23] and Cactus [24] mostly contained computational kernels with specific properties, e.g., irregularity; Altis [31] was designed for modern GPU architectures and runtime environments; Chai [25] and Hetero-Mark [26] captured the collaboration between CPUs and GPUs; and DeepBench [27], MLPerf [28], Tango [29], and TBD [30] were to characterize the behavior of neural networks on GPUs. Although some of these benchmarks contain several applications with data sharing, they are heavily optimized to avoid or minimize synchronization across threads, making it hard for the comprehensive evaluation and analysis of fine-grained synchronization on GPUs.

HeteroSync [32] focused on benchmarking the synchronization effects by combining a set of GPU microbenchmarks with different kinds of synchronization primitives. As its main target was to characterize the scalability of synchronization primitives for different coherence protocols and consistency models on integrated CPU-GPU systems, it employed small-scale microbenchmarks (e.g., mutexes and semaphores with limited parallelism) rich in atomic operations and focused on low-level metrics, such as the cycle count of individual synchronization primitives. Different from HeteroSync, our GPUSync benchmark targets fine-grained synchronization scenarios and architectural support exploration in discrete GPU architectures. It consists of eight representative applications spanning diverse algorithms and domains. These applications exhibit substantially more complex synchronization behaviors and are significantly longer-running than microbenchmarks in HeteroSync. By doing so, we emulate the varied synchronization behaviors in real-world workloads.

Many other studies have focused on creating synchronization benchmarks in the conventional multi-core CPU architectures, with both locking [34] and TM [35–39] implementations. However, due to the distinct architectural characteristics of GPUs compared to CPUs, many of these benchmarks are not suitable for GPU computing. Those that are suitable cannot be directly deployed on GPUs, and significant effort would be required for adaptation. Besides, comprehensive evaluations for fine-grained synchronization tailored for GPU architectures are imperative.

4 Fine-grained synchronization applications on GPUs (GPUSync)

At present, there are few GPU benchmark suites for fine-grained synchronization that cover a wide enough range of algorithms and application domains, or fully stress a broad spectrum of synchronization scenarios. GPUSync is the first benchmark designed to satisfy all these requirements.

4.1 Applications

GPUSync currently consists of eight applications: DMR, SmallBank (SBank), TPCC, kmeans (KM), labyrinth (LB), genome (GN), hash table (HT) and random array (RA). Among them, DMR is a popular scientific application used in many areas such as the numerical solution of partial differential equations and graphics [40]. In this article, we modify the DMR implementation from the LonestarGPU benchmark suite [21], in which the accesses

to shared data are avoided via a 3-phase conflict resolution scheme [19]. To validate its effectiveness, we compare our implementation against LonestarGPU on an RTX 3080 GPU. The results show that the performance of both implementations is comparable, with a mere 2% difference across all inputs.

SBank and TPCC are two typical OLTP workloads that are widely used in credit card transactions, banking, and stock markets. These applications are characterized by high parallelism and numerous concurrent transactions, making them well-suited for GPU computing [4, 41]. In this article, we port them from the DrTM implementation [42], one of the fastest distributed transaction systems on CPUs. We compare our implementations with DrTM and observe significant speedups: 4.5× for SBank and 2.4× for TPCC on an RTX 3080 GPU across all inputs.

KM, LB, and GN are three classic benchmarking workloads that are widely used in data mining, engineering, and bioinformatics domains, respectively. Their computational characteristics are suitable for GPU computing, and they are included in several GPU benchmark suites [18,31]. To fully exhibit their synchronization behavior, we port them from STAMP [37], one of the most popular TM benchmark suites for multi-core CPUs, which is a common approach in fine-grained synchronization researchas on GPUs [13, 14, 43].

HT is comprised of a basic indexing data structure, which is suitable for GPU computing with massive parallelism [44], and RA is a micro workload. Compared with the other applications in GPUSync, they are relatively simple and can help identify some specific system features on GPUs. For example, branch divergences caused by fine-grained synchronization primitives and synchronization failures can affect the performance. To precisely identify the frequency of these branch divergences and their impact on performance, HT and RA are necessary with the configuration guaranteeing there is no other branch divergence during the execution.

(1) *DMR.* DMR takes an unrefined Delaunay mesh as input and produces a new mesh that satisfies some specific constraints (e.g., no angle in the mesh is less than 30 degrees). For a given Delaunay mesh, the refinement is accomplished by iteratively processing the triangles that do not satisfy the constraints.

The massive data parallelism of DMR exists in the work list of triangles, which is suitable to be accelerated on GPUs [19]. The refinement for triangles whose cavities (the region of the influenced triangles around a triangle is called the cavity of the triangle) do not overlap is completely independent, thus can be performed in parallel. However, if the cavities of two triangles overlap with each other, which we call a conflict, only one of them can be processed at a time. Therefore, it requires fine-grained synchronization for each to be refined triangles and the ones in its cavity, to resolve data access conflicts across threads in the parallelization of DMR.

(2) *SBank.* SBank [45] models a simple banking application. It consists of six types of transactions for inquiring and modifying accounts that are stored in three database tables. Transactions send-payment (SP), amalgamate (AMG), deposit-checking (DC), write-check (WC) and transact-savings (TS) perform read-write operations on databases, and transaction balance (BAL) performs read-only operations on databases. The three database tables utilize hash tables to store the information for users, checking, and saving accounts, respectively.

Lots of transactions can be performed simultaneously among the threads. In case multiple transactions operate on the same account, only one of them can be processed at a time. Thus, it requires fine-grained synchronization for each account to resolve data access conflicts across threads.

(3) *TPCC.* TPCC simulates a warehouse-centric order processing application with five different types of transactions. Transactions new-order (NEW), payment (PAY), and delivery (DLY) perform read-write operations on databases, and transactions order-status (OS) and stock-level (SL) perform read-only operations on databases. In TPCC, there are tens of database tables to store the information of guest accounts, guest orders, and product details. Some of these tables employ hash tables for rapid lookup, while others leverage b+ trees to support range searches.

Unlike the SBank application with simple transactions, TPCC features large and complex transactions that involve multiple insertions and deletions to databases. To reduce the chance of conflicts, we leverage transaction chopping with optimizations in DrTM [42], and manually optimize the fine-grained synchronization behavior for all transactions in our GPU implementation.

(4) *KM.* KM groups objects in an N-dimensional space into K clusters. It is suitable for GPU computing with the massive objects being grouped simultaneously among all the threads. Since each object needs to update its respective cluster center after determining the belonging cluster, fine-grained synchronization is needed for each cluster, to protect the updates to the cluster centers.

(5) *LB.* LB implements a variant of Lee's routing algorithm [46], which finds non-overlapping routes between pairs of points on a grid. The routes between different pairs of points can be searched simultaneously. As such, it is suitable for GPU computing with lots of routes being searched simultaneously. However, the routes of different pairs of points may overlap with each other. To ensure that any two routes do not overlap, fine-grained synchronization is needed for the maze grid points in a route when adding this route to the global grid.

Due to the multitude of potential routes between each pair, a TB with a large number of parallel threads is

responsible for searching all potential routes between a pair, and then performs the fine-grained synchronization. It is different from other GPUSync applications, in which fine-grained synchronization is performed among individual threads.

(6) *GN*. GN implements a gene assembler that takes a large number of DNA segments and matches them to reconstruct the original source genome. With the large number of DNA segments, GN is suitable for GPU computing. The process in GN consists of two steps: hashing all segments to find unique segments, and matching the unique segments to reconstruct the original genome.

In the first step, segments are hashed simultaneously among the threads. To guarantee the uniqueness of segments within hash tables, fine-grained synchronization is required for each hash bucket, to protect concurrent accesses to the hash tables. In the second step, the matched segments are searched simultaneously among the threads. To ensure the searching correctness, fine-grained synchronization is needed for each unmatched segment, to protect the concurrent accesses to them.

(7) *HT*. HT operates on a chained hash table that is shared among all threads. Three types of operations are involved in HT: search, insertion, and deletion. The first type is read-only, while the other two are read-write operations. To show various usage scenarios of fine-grained synchronization, multiple operations are batched into a single transaction in HT, which must be processed atomically. Since operations are executed simultaneously among all the threads, fine-grained synchronization is needed for each bucket or key-value pair, to protect data accesses to the hash table.

(8) *RA*. In the RA microbenchmark, each thread randomly accesses (reads from/writes to) one or multiple locations of a shared data array. To guarantee correctness and efficiency, fine-grained synchronization is required for each location in the array when accessing the data in the location.

4.2 Configurations and data sets

To cover a wide range of synchronization scenarios, various sets of configurations and input data should be provided. As shown in Table 2, all the applications in GPUSync are equipped with diverse parameters. By doing so, GPUSync can exhibit different synchronization behaviors, including the amount of shared data, number of parallel threads, read and write numbers for shared data per critical section/transaction, working set sizes, critical section/transaction lengths, sizes of read and write sets for shared data per transaction, and the percentage of time spent in synchronization.

However, (1) these diverse synchronization behaviors and (2) wide-ranging parameter values for each synchronization behavior on GPUs create an exponentially large configuration space, presenting a significant challenge for comprehensive evaluation. To tackle this challenge, we initially analyze all synchronization behaviors thoroughly and explore their core nature. From this analysis, we identify and categorize them into four key features. The detailed analysis is listed in Table 3. Additionally, we provide a description for each feature using either quantitative or qualitative methods, thereby facilitating the evaluation.

Contention degree. It is a critical aspect of synchronization characteristics, as it directly reflects the possibility of a thread successfully accessing the shared data. As shown in Table 3, the contention degree is determined conjointly by these synchronization behaviors: amount of shared data, number of parallel threads, and the read and write numbers for shared data per critical section/transaction. To facilitate the synchronization characterization and configuration setting, it is imperative to ensure a comprehensive yet non-redundant coverage of synchronization execution cases. To achieve this, we explore quantifying the contention degree using a unified metric based on the synchronization failure ratio, as it directly reflects the degree of contention. However, because the actual synchronization failure ratio is an empirical measure that can be affected by specific synchronization mechanisms and hardware limitations, we instead introduce the ideal minimal synchronization failure ratio as the quantification metric. It is defined as the ratio of the minimal number of synchronization failures to the total synchronization attempts. In this article, we name this quantification as CQ, the abbreviation of contention degree quantification.

However, identifying the minimal number of synchronization failures is an NP-hard problem. In the context of fine-grained synchronization on GPUs, significant time would be consumed with the large number of parallel threads and extensive read and write numbers for shared data per critical section/transaction. As such, we utilize a greedy algorithm to approximate the solution. This approach selects the thread with the largest non-conflicting thread set among all the unexecuted threads for each step. Using this algorithm, we simulate the synchronization procedure repeatedly with the related synchronization behaviors. From these simulations, we derive the average near-ideal minimal synchronization failure ratio, which we designate as the CQ value. The simulation code is available in GPUSync.

Table 2 Recommended configurations and data sets for GPUSync. "SMs" in the second column refers to the number of SMs in the GPU where the application is executed. In this article, the value is set to 24.

Application	Parameters	Description
DMR-1	-g SMs -b 512 -n 250K	The refinement is performed iteratively on inputfile n. The inputfile consists of n nodes. The parallel thread number is $g \times b$.
DMR-2	-g SMs -b 256 -n 500K	
DMR-3	-g SMs -b 512 -n 1M	
DMR-4	-g SMs -b 512 -n 5M	
SBank-1	-g SMs -b 512 -n 200000 -nh 8000 -ops 4000000 -w 25, 15, 15, 15, 15, 15	The number of user accounts and hot user accounts is n and nh respectively. There are ops transactions with a distribution ratio w. The parallel thread number is $g \times b$.
SBank-2	-g SMs -b 512 -n 800000 -nh 32000 -ops 4000000 -w 25, 15, 15, 15, 15, 15	
SBank-3	-g SMs -b 256 -n 800000 -nh 32000 -ops 4000000 -w 2, 2, 2, 2, 2, 90	
TPCC-1	-g SMs -b 256 -ops 4000000 -w 45, 43, 4, 4, 4	There are ops transactions with a distribution ratio w. The parallel thread number is $g \times b$.
TPCC-2	-g SMs -b 256 -ops 4000000 -w 4, 3, 3, 45, 45	
KM-1	-g SMs -b 32 -m 64 -n 64 -t 0.05 -i random-n2048-d16-c16.txt	The number of cluster centers is varied from m to n. A convergence threshold of t is used, and analysis is performed on inputfile i. The inputfile consists of n points of d dimensions generated about c centers. The parallel thread number is $g \times b$.
KM-2	-g SMs -b 32 -m 64 -n 64 -t 0.05 -i random-n16384-d16-c16.txt	
KM-3	-g 10 -b 128 -m 1000 -n 1000 -t 0.05 -i random-n16384-d16-c16.txt	
KM-4	-g 10 -b 128 -m 1000 -n 1000 -t 0.05 -i random-n65536-d32-c16.txt	
LB-1	-g 512 -b 256 -i random-x32-y32-z3-n64.txt	The inputfile i consists of a maze of dimensions $x \times y \times z$. n paths are routed. The parallel thread number is $g \times b$.
LB-2	-g 512 -b 256 -i random-x64-y64-z3-n48.txt	
LB-3	-g 512 -b 256 -i random-x128-y128-z3-n64.txt	
LB-4	-g 512 -b 256 -i random-x256-y256-z3-n256.txt	
GN-1	-g0 SMs -b0 128 -g1 SMs -b1 512 -gn 1024 -s 32 -n 32768	Gene segments of s nucleotides are sampled from a gene with gn nucleotides. A total of n segments are analyzed to reconstruct the original gene. The parallel thread number is $g \times b$.
GN-2	-g0 SMs -b0 128 -g1 SMs -b1 512 -gn 2048 -s 32 -n 32768	
GN-3	-g0 SMs -b0 512 -g1 SMs -b1 512 -gn 16384 -s 64 -n 16777216	
HT-1	-g SMs -b 256 -z 1 -nn 24K -nb 6K -op 2 -top 1M -w 0, 100, 0	The hash table consists of nb buckets and supports storing keys with a value from 0 to $nn-1$. There are top transactions with a distribution ratio w, and each transaction consists of op operations. The zipfian distribution of keys to be inserted/searched/deleted is enabled with z. The parallel thread number is $g \times b$.
HT-2	-g SMs -b 256 -z 0 -nn 24K -nb 6K -op 4 -top 1M -w 0, 100, 0	
HT-3	-g SMs -b 256 -z 0 -nn 24K -nb 6K -op 4 -top 1M -w 96, 2, 2	
HT-4	-g SMs -b 1024 -z 0 -nn 32M -nb 8M -op 1 -top 1M -w 50, 25, 25	
RA-1	-g SMs -b 256 -z 0 -n 6K -op 4 -top 1M -branch 0 -w 0, 100, 0	The data array length is n. There are top transactions with a distribution ratio w, and each transaction consists of op operations. The zipfian distribution of data to be inserted/searched/deleted is enabled with z. The parallel thread number is $g \times b$.
RA-2	-g SMs -b 256 -z 0 -n 6K -op 4 -top 1M -branch 0 -w 96, 2, 2	
RA-3	-g SMs -b 256 -z 0 -n 6K -op 2 -top 1M -branch 0 -w 0, 100, 0	
RA-4	-g SMs -b 1024 -z 0 -n 512M -op 2 -top 1M -branch 0 -w 50, 25, 25	

To validate our simulation, we calculate the true minimal synchronization failure ratio by exhaustively enumerating all possible scenarios in small-scale cases. The parameters for these cases are set with thread numbers ranging from 2 to 8, shared data from 1 to 16, and reads/writes per critical section/transaction from 1 to 4. The results show an average difference of only 3.6% between our approach and the true minimum. The maximum difference observed is 13.4%, under a specific configuration with 2 shared data, 7 threads, and 1 read/write. These results validate the reliability of our greedy approximation algorithm.

Critical section/transaction executing time. It is another key feature of synchronization characteristics, determining the waiting period a failed thread must endure before it accesses the shared data successfully. As detailed in Table 3, the executing time is determined by the following synchronization behaviors: number of parallel threads, working set, and critical section/transaction length.

Since GPUs utilize a vast number of parallel threads and fast context switching among active warps to mitigate memory access latencies, the impact of the (1) number of parallel threads and (2) working set on executing time is highly dependent on the specific GPUs, which have distinct computing and memory capabilities and employ diverse warp scheduling strategies. Moreover, estimating this influence theoretically is impractical. Instead, we describe the critical section/transaction executing time qualitatively as tiny, short, medium, and long, based on the length of the critical section/transaction that can directly reflect the executing time.

Read and write sets for shared data per transaction. In the locking mechanism, a single lock can protect multiple shared data elements when they are always accessed simultaneously by threads. In contrast, TM

Table 3 Analyses of the synchronization behaviors.

Synchronization behaviors	Related parameters of GPUSync applications				The key synchronization features
	DMR; LB;	SBank; GN;	TPCC; HT;	KM; RA	
Amount of shared data	n; i;	n, nh; gn, s;	–; nb;	m, n; n	*Contention degree* A smaller number of shared data suggests a more intense level of contention.
Number of parallel threads	g, b; g, b;	g, b; g0, b0, g1, b1;	g, b; g, b;	g, b; g, b	*Contention degree* The larger the number of parallel threads, the more threads compete for shared data, indicating a more intense level of contention. *Critical section/transaction executing time* As parallel threads contend for execution units, a higher number of threads leads to increased critical section/transaction executing time.
Read and write numbers for shared data per critical section/transaction	–; i;	w; –;	w; op, w;	–; op, w	*Contention degree* As the read/write failure to any shared data within a critical section/transaction results in the failure of the entire critical section/transaction, larger values result in a more intense level of contention.
Working set	n; i;	–; gn, s, n;	–; nn;	m, n, i; n	*Critical section/transaction executing time* Warps stall when they cannot overlap memory access latencies through context switching. Thus, a larger working set size, which may incur longer memory access latencies, results in a longer execution time.
Critical section/transaction length	–; i;	w; –;	w; op, w;	i; op, w	*Critical section/transaction executing time* A larger critical section/transaction length indicates a longer execution time.
Read and write sets for shared data per transaction	–; i;	w; –;	w; op, w;	i; op, w	*Read and write sets for shared data per transaction*
Percentage of time spent in synchronization	–; i;	w; s;	w; op, w;	i; op, w	*Percentage of time spent in synchronization*

encapsulates each read and write operation to shared data within a transactional primitive, to ensure that either all of them appear to execute or none of them does. As a result, the collective read and write sets within a transaction in TM systems, which stem from the read and write operations to shared data, may significantly exceed the read and write numbers per critical section/transaction.

To this end, we present the read and write sets for shared data per transaction as another distinct feature of synchronization characteristics. It is used to stress the TM designs, because the overhead brought by the TM systems (especially the hardware TM systems) is much related to these sets. On GPUs, with their massively parallel executing threads, this overhead can be substantial, significantly impacting performance.

Percentage of time spent in synchronization. It does not directly influence performance. Instead, it represents how important synchronization is to the whole application, helping estimate potential performance gains from different synchronization schemes.

The above categorization of synchronization behaviors helps guide the configuration setup. Among these, the first three features directly influence application performance (e.g., execution time). Specifically, contention degree and critical section/transaction execution time jointly affect the synchronization failure ratio. This is because the former reflects the possibility of access conflicts, while the latter determines the duration of delays when conflicts occur. The read and write sets for shared data per transaction indicate the storage overhead of a synchronization scheme, which also significantly impacts performance. Consequently, this metric helps determine whether a synchronization scheme can maintain high performance under varying storage demands, making it useful to stress different proposals.

Based on the four features, we design comprehensive evaluation parameters for the GPUSync applications to cover a wide range of synchronization scenarios, and list them in Table 2. Applications with lower numbers have a smaller computation scale relative to those with higher numbers.

4.3 Implementations

To enlarge the applicability of GPUSync to different synchronization schemes, we implement all the applications using two of the most common synchronization techniques, locking and TM. The only difference between the locking and TM implementation for the GPUSync applications is the synchronization method they use. Currently, GPUSync is implemented using CUDA 12, to better support modern GPU architectures and modern GPU runtimes (e.g., the

APIs of Lock	APIs of TM
1: **Lock initialization:**	1: **TM initialization:**
2: __host__ bool init_gpu locks (uint num_locks)	2: __host__ bool TXNew (unit num_threads)
3: **Lock deallocation:**	3: **TM deallocation:**
4: __host__ void destroy_gpu_locks ()	4: __host__ void TXFree ()
5: **Mutual exclusive lock:**	5: **Transaction Primitives:**
6: __device__ bool lock (Address addr)	6: __device__ void TXBegin ()
7: __device__ void unlock (Address addr)	7: __device__ Value TXRead (Address addr)
	8: __device__ void TXWrite (Address addr, Value val)
	9: __device__ bool TXCommit ()

Figure 4 APIs of locking and TM systems in GPUSync.

independent thread scheduling in modern GPUs). Additionally, we plan to provide the OpenCL implementation in the future to further extend the portability of GPUSync.

In GPUSync, we encapsulate synchronization schemes into a set of standardized interfaces (APIs) to establish a unified platform that supports diverse synchronization mechanisms, as shown in Figure 4. Applications in GPUSync fulfill their synchronization demands by invoking these common APIs. These APIs are consistent with prior research [1–15]. By doing so, developers can integrate their synchronization schemes into GPUSync easily.

5 Usage methodology of GPUSync

To demonstrate the usefulness of GPUSync for GPU fine-grained synchronization research, we run it with a classical locking and TM system across an old and a modern GPU respectively, and conduct a thorough performance analysis tailored specifically for GPU architectures.

5.1 GPUSync running

To demonstrate the portability and practical usefulness of GPUSync with different synchronization techniques, we run it with a classical locking (TLLL [1]) and software TM (GPU-STM [13]) mechanism, respectively. TLLL [1] implements mutex locks on GPUs with atomic primitives, which introduces a lock stealing algorithm to avoid live locks. It is compatible with modern GPUs featuring independent thread scheduling. Unlike it, GPU-STM [13] encounters indeterminate errors on modern GPUs with independent thread scheduling. This occurs because GPU-STM employs the SIMT execution paradigm to realize hierarchical validation, which is designed to mitigate the TM overhead to support the massive multithreading of GPUs.

Migrating GPU-STM to the modern GPUs is more complex than initially thought. Given the commonality of this challenge, we provide a detailed discussion for reference. Figure 5(a) lists the pseudo code for the transaction implementation of GN-① (the first kernel of GN with fine-grained synchronization), which is very common for applications with fine-grained synchronization on GPUs. When using GPU-STM, threads within a warp must converge at two specific points (lines 7 and 24 in Figure 5(a)). There are also two converging points (lines 10 and 21 in Figure 5(b)) in the transaction committing primitive STM_END() (line 21 in Figure 5(a)). While this is inherent to the SIMT execution paradigm, it poses difficulties when executed correctly with the independent thread scheduling feature.

Firstly, as illustrated in the CUDA documentation, the active threads at the converging point must be identified accurately to avoid undefined behavior. Thus, a __ballot_sync() primitive is required to get the identifications of active threads for each branch (conditional and loop branch) prior to the point. For example, to make the active threads converge at line 10 in Figure 5(b), the above primitive needs to be applied before lines 6–8 in Figure 5(b), and lines 6 and 19 in Figure 5(a). This is very complex, easily resulting in deadlocks if not processed with caution.

Furthermore, we observe system crashes when threads following different control flow paths within a warp attempt simultaneous synchronization at distinct points using CUDA primitives, as illustrated in Figure 5(a). It causes waiting threads to continue the execution before all active threads reach the synchronization point. This issue may stem from these primitives sharing the same hardware barrier, which leads to inaccuracies in counting the number of arrived threads. To address this, programmers have to use alternative methods to ensure convergence. In our implementation, we utilize the approach depicted in Figure 5(c) to realize explicit convergence at line 24 in Figure 5(a) to avoid these errors.

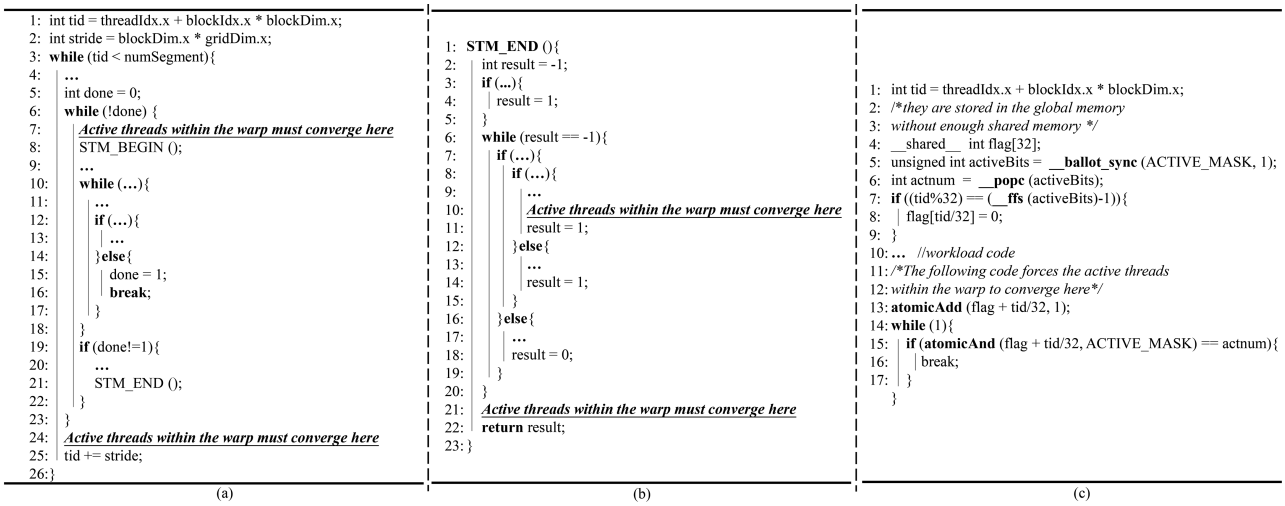

Figure 5 (a) Pseudo code for the transaction of GN-①; (b) pseudo code for transaction committing primitive in GPU-STM; (c) explicit implementation of warp-level convergence.

Table 4 The main differences of the TITAN X and RTX 3080 GPU.

GPUs	Number of SMs	L1 cache size (kB)	L2 cache size (MB)	Atomic units	Warp schedulers	Register file size per SM (kB)	Shared memory per SM (kB)	Global memory size (GB)	Max clock rate (GHz)	Memory clock rate (MHz)
TITAN X	24	48	3	16	4	256	96	12	1.08	3505
RTX 3080	70	128	6	16	4	256	128	12	1.78	9501

5.2 Analyzing methodology

To identify how the unique GPU characteristics impact the execution efficiency of fine-grained synchronization, we perform a comprehensive evaluation and systematically analyze performance metrics. Specifically, we evaluate the performance and synchronization failure ratio, and profile applications using Nsight Compute and NVProf tools. After analyzing all the metrics collected by these tools, we focus on the following metrics.

(1) *The active threads per warp (avg_acthreads_perwarp).* It represents the average number of active threads within a warp that are being executed. Ideally, it is desirable for all 32 threads in a warp being active. However, factors such as branch divergences and the use of warp-level barrier primitives can result in only a subset of threads being active, leading to reduced execution efficiency. This metric provides insight into the actual parallelism during execution (much related to the synchronization failure ratio) and frequency of branch divergences in fine-grained synchronization mechanisms with minimal warp-level barriers.

(2) *The average number of eligible warps per cycle (avg_ewarps_percycle).* It represents the number of warps that are ready to issue their next instruction per cycle. Factors such as memory accesses and the use of warp-level and TB-level barrier primitives can cause warps to enter a wait (i.e., noneligible) state, decreasing the value of avg_ewarps_percycle and negatively impacting execution efficiency. Besides, this metric can also be used to indicate the actual parallelism during execution, which shows the maximum number of warps executed simultaneously per cycle.

(3) *The main warp stalling reason (main_warp_stall_reason).* This metric can explain why warps are not being scheduled to execute instructions. It is used to gain a deep understanding of the underlying issues behind the low execution efficiency and identify the performance bottleneck.

6 Evaluation

In this section, we provide a quantitative analysis of the synchronization characteristics for GPUSync applications. To explore the programmability and performance significance of independent thread scheduling in modern GPUs on fine-grained synchronization, we conduct the experiments on two different GPUs: NVIDIA TITAN X and RTX 3080. The TITAN X is based on the Maxwell microarchitecture, which does not feature independent thread scheduling. In contrast, the RTX 3080 utilizes the Ampere microarchitecture, including this capability. As shown in Table 4, the RTX 3080 demonstrates significant architectural improvements over the TITAN X. It features a substantially higher number of SMs, along with dramatically increased core and memory clock rates. The RTX 3080 is also equipped

Table 5 The basic characterization of the GPUSync applications. Metrics (with abbreviations) include: read and write numbers for shared data per critical section/transaction (R_N and W_N), amount of shared data (SD_N), number of parallel threads (P_N), contention degree quantification (CQ), the size of working set (WK_S), the critical section/transaction length (LEN), critical section/transaction executing time (E_T), average sizes of read and write sets per transaction (R_S and W_S), and percentage of time spent in synchronization (P_T). ① in this table refers to the first kernel of GN with fine-grained synchronization. ② in this table refers to the second kernel of GN with fine-grained synchronization.

Applications	Contention degree				Critical section/transaction executing time			Read and write set for shared data per transaction	Percentage of time spent in synchronization
	R_N/ W_N	SD_N	P_N (k)	CQ	WK_S (MB)	LEN	E_T	R_S/W_S (B)	P_T (%)
DMR-1	0/7	250k	12	50	272	18059	Long	132/40	100
DMR-2		500k	6	20	495			132/40	100
DMR-3		1M	12	20	991			132/40	100
DMR-4		5M	12	10	4959			132/40	100
SBank-1	1/2	53k	12	50	48	7	Tiny	8/8	41
SBank-2	1/2	213k	12	20	48	7		8/8	40
SBank-3	2/1	213k	6	10	48	3		8/4	36
TPCC-1	2/2	10k	6	70	1405.9	4116	Medium	120/120	33
TPCC-2	3/1	10k	6	60	488.5	2158		20/36	25
KM-1	0/1	64	0.75	90	0.2	173	Short	68/68	5
KM-2		64	0.75	90	1.2			68/68	5
KM-3		1000	1.25	50	1.3			68/68	2
KM-4		1000	1.25	50	9			132/132	25
LB-1	0/25	3072	512	100	6	365	Short	104/100	1
LB-2	0/41	12288	512	90	24.1			180/180	15
LB-3	0/83	49152	512	90	96.2			380/376	3
LB-4	0/173	196608	512	90	384.8			860/812	9
GN-1	① 0/1 ② 0/1	① 1k ② 31k	① 3 ② 12	① 70 ② 20	① 1.1 ② 1.3	① 2061 ② 222	① Medium ② Short	① 4/4 ② 4/4	① 58 ② 63
GN-2		① 2k ② 62k	① 3 ② 12	① 50 ② 10	① 1.1 ② 1.5	① 2061 ② 222			① 58 ② 63
GN-3		① 16k ② 1008k	① 12 ② 12	① 30 ② 0	① 1088.1 ② 1032.1	① 3757 ② 222			① 59 ② 64
HT-1	0/2	6k	6	70	0.7	62	Tiny	24/16	61
HT-2	0/4	6k	6	90	0.7	62		108/32	67
HT-3	4/1	6k	6	80	0.6	4		144/4	57
HT-4	1/1	8M	24	0	730.2	25		28/4	58
RA-1	0/4	6k	6	90	0.02	40	Tiny	0/16	86
RA-2	4/1	6k	6	80	0.02	24		16/4	84
RA-3	0/2	6k	6	70	0.02	20		0/8	80
RA-4	2/1	512M	24	0	2048	14		8/4	84

with larger L1 and L2 caches, and more shared memory per SM. To make the comparison as fair as possible, we configure all GPUSync applications to run on 24 SMs on the RTX 3080, matching the number available on the TITAN X.

6.1 GPUSync basic characteristics

Table 5 presents the basic synchronization characteristics of applications in GPUSync. All the contention degree, critical section/transaction executing time, sizes of read and write sets for shared data per transaction, and percentage of time spent in synchronization vary at different orders of magnitude, thus showing that GPUSync covers many different synchronization execution scenarios.

For the contention degree, we compute the CQ value. As shown in Table 5, different degrees of contention are offered for each application, and a wide range of contention degrees is covered in GPUSync, ranging from nearly 0 to nearly 100.

As introduced in Subsection 4.2, we describe the critical section/transaction executing time as tiny, short, medium, and long. Given that the number of parallel threads is specified in the contention degree column in Table 5, we focus on introducing the size of the working set and the critical section/transaction length in this part. As shown in Table 5, GPUSync comprises of (1) one long application (DMR) with tens of thousands of PTX instructions per transaction; (2) two medium applications (TPCC and kernel ① of GN) with thousands of PTX instructions per transaction; (3) three short applications (KM, LB and kernel ② of GN) with hundreds of PTX instructions per

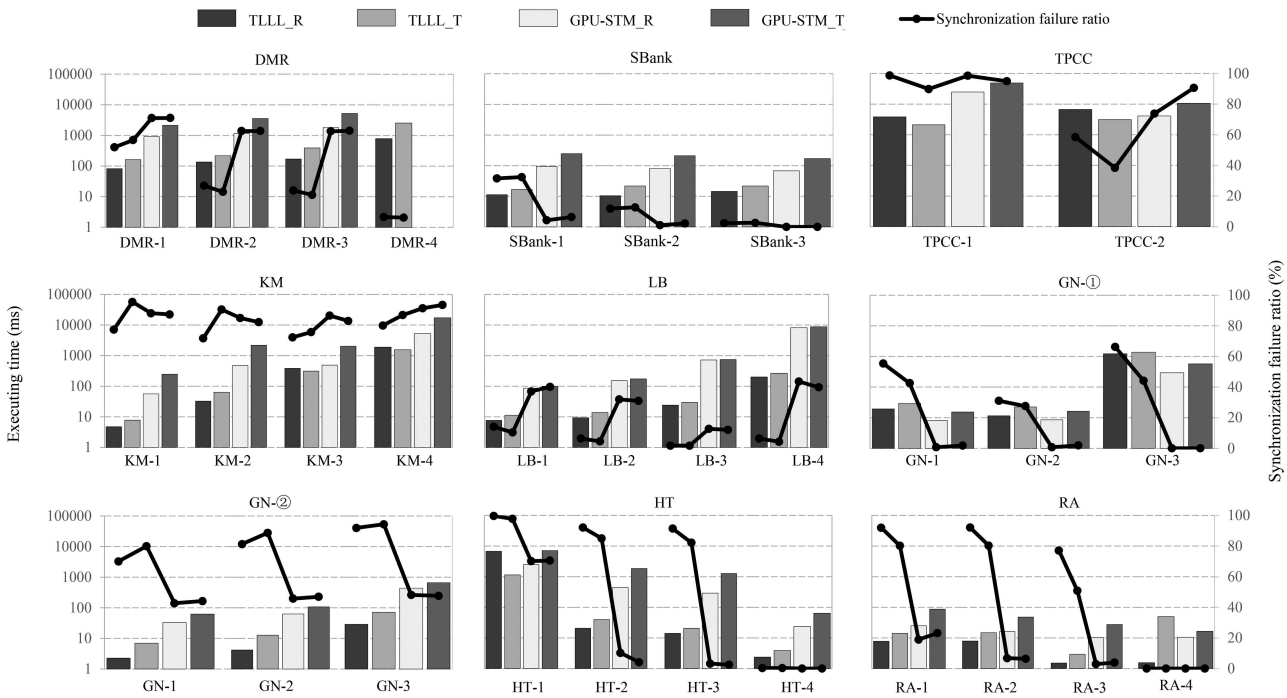

Figure 6 Performance and synchronization failure ratio of applications in GPUSync.

transaction; and (4) three tiny applications (SBank, HT and RA) with only a few PTX instructions per transaction.

The average sizes of read and write sets per transaction are evaluated by running GPUSync repeatedly with the GPU-STM. They are given as a guide to TM system designers for sizing their transactional buffers to handle the common case. As shown in Table 5, various sizes of read and write sets per transaction are covered in GPUSync. Notably, while LB with LB-4 exhibits the largest sets, DMR costs the greatest storage for the sets, due to the massive number of parallel threads that process transactions independently.

At last, we show the percentage of time spent in synchronization in Table 5 to identify how important the synchronization is to the whole GPU kernel.

6.2 Performance analysis

As shown in Figure 6 and Table 6, TLLL_R and GPU-STM_R denote the implementation using TLLL and GPU-STM on the RTX 3080, respectively, while TLLL_T and GPU-STM_T represent their respective implementations on the TITAN X. The evaluation results provide four findings that are consistent with the conventional wisdom.

(1) Given the RTX 3080's significant architectural improvements over the TITAN X (shown in Table 4), most applications exhibit superior performance compared to the TITAN X.

(2) TLLL outperforms GPU-STM for many applications with the common sense that, although TM is typically more user-friendly, it results in greater overhead than locking mechanisms.

(3) TLLL shows a higher synchronization failure ratio compared to GPU-STM for many GPUSync applications. As evidenced by the avg_acthreads_perwarp and avg_ewarps_percycle in Table 6, TLLL exhibits higher actual parallelism during execution. This results in a higher number of threads competing for shared data, thereby increasing the potential for synchronization conflicts.

(4) The avg_ewarps_percycle values are low across all GPUSync applications, indicating many warps are in a wait state for each cycle. Besides, waiting for memory accesses is the main warp stalling reason for many applications.

(5) On the RTX 3080, the avg_acthreads_perwarp values for TLLL are smaller or comparable to those on the TITAN X. The independent thread scheduling feature makes a warp become eligible if any thread within it is ready to execute the next instruction. This can increase the number of eligible warps but may decrease avg_acthreads_perwarp values. Moreover, since GPU-STM requires all threads within a warp to converge at the start and end of a transaction, the avg_acthreads_perwarp values are similar on the two GPUs.

Furthermore, our evaluation results reveal several instances that contradict the above generalizations.

DMR. Due to the significant storage overhead associated with GPU-STM, DMR-4 is unable to execute successfully on either GPU because the memory requirements exceed the 12 GB capacity of the device. Furthermore, as shown

Table 6 Key performance metrics collected from the NVIDIA Nsight Compute and NVProf. ① in this table refers to the first kernel of GN with fine-grained synchronization. ② in this table refers to the second kernel of GN with fine-grained synchronization. # denotes the main warp stalling reason being the barrier. * denotes the main warp stalling reason being wait. The ones without # or * in "Main_warp_stall_reason" denote the main warp stalling reason being waiting for memory accesses.

	Avg_acthreads_perwarp				Avg_ewarps_percycle				Main_warp_stall_reason (%)			
	TLLL_R	TLLL_T	GPU-STM_R	GPU-STM_T	TLLL_R	TLLL_T	GPU-STM_R	GPU-STM_T	TLLL_R	TLLL_T	GPU-STM_R	GPU-STM_T
DMR-1	3.0	14.7	3.2	2.2	0.8	0.2	0.5	0.3	63	85	61	69
DMR-2	3.2	15.5	3.5	2.7	0.4	0.2	0.3	0.2	62	82	63	71
DMR-3	3.2	17.5	3.2	2.6	0.8	0.2	0.5	0.3	65	88	65	73
DMR-4	3.3	9.0	–	–	0.8	0.1	–	–	65	94	–	–
SBank-1	7.4	17.3	4.8	6.2	0.7	0.4	0.5	0.3	65	73	52#	49#
SBank-2	11.5	19.7	5.1	6.9	0.5	0.3	0.5	0.3	75	86	49#	46#
SBank-3	10.8	15.2	6.1	8.0	0.3	0.3	0.3	0.2	70	76	31	44
TPCC-1	6.2	5.9	4.7	3.2	0.4	0.3	0.2	0.2	56	66	67	62
TPCC-2	2.1	18.7	10.7	11.7	0.6	0.2	0.2	0.2	60	71	51	49
KM-1	5.2	16.4	5.3	5.7	0.6	0.03	0.3	0.03	49	77	61	79
KM-2	5.2	17.9	5.4	5.0	0.6	0.03	0.3	0.03	41	77	59	79
KM-3	17.5	31.3	9.6	12.0	0.5	0.1	0.2	0.1	50	82	55	68
KM-4	19.3	31.5	7.0	12.0	0.2	0.1	0.2	0.1	49	86	54	72
LB-1	15.8	16.2	12.3	13.7	0.9	0.5	0.2	0.2	42#	46#	67#	62#
LB-2	18.7	18.8	18.3	16.4	1.3	0.5	0.3	0.2	29#	35#	48#	55#
LB-3	21.4	21.7	19.5	19.1	0.9	0.6	0.3	0.3	35	45	42*	37*
LB-4	23.4	25.0	24.5	23.7	0.7	0.8	0.4	0.4	64	65	50*	45*
GN-1	① 3.8 ② 13.0	① 3.8 ② 16.6	① 8.8 ② 7.7	① 10.1 ② 9.9	① 0.2 ② 0.8	① 0.2 ② 0.5	① 0.2 ② 0.7	① 0.2 ② 0.5	① 81* ② 71*	① 70* ② 71*	① 55* ② 38*	① 51* ② 34*
GN-2	① 3.9 ② 12.7	① 4.7 ② 16.7	① 8.9 ② 10.0	① 9.8 ② 15.3	① 0.2 ② 0.8	① 0.2 ② 0.5	① 0.2 ② 0.6	① 0.2 ② 0.5	① 80* ② 73*	① 69* ② 73*	① 58* ② 45*	① 51* ② 42*
GN-3	① 3.8 ② 13.4	① 5.5 ② 16.2	① 15.1 ② 11.4	① 15.6 ② 24.5	① 0.9 ② 0.8	① 0.7 ② 0.5	① 0.9 ② 0.6	① 0.7 ② 0.5	① 79* ② 73*	① 67* ② 73*	① 76* ② 50*	① 63* ② 44*
HT-1	19.6	1.4	5.5	2.4	0.2	0.3	0.4	0.2	83	47	69	78
HT-2	3.3	5.6	3.1	3.0	0.7	0.3	0.4	0.2	54	67	66	81
HT-3	4.1	9.5	3.5	3.7	0.7	0.3	0.5	0.2	51	66	65	80
HT-4	21.5	27.2	1.9	6.5	0.5	0.2	1.4	0.5	74	82	47#	22#
RA-1	7.4	3.9	6.9	5.9	0.4	0.2	0.4	0.2	69	70	61	78
RA-2	7.5	3.9	6.9	9.1	0.4	0.2	0.4	0.2	70	73	52	75
RA-3	11.4	6.4	9.4	8.9	0.4	0.2	0.4	0.2	69	70	55	75
RA-4	31.7	31.9	6.5	8.8	0.04	0.002	1.4	0.5	99	92	48#	32#

in Figure 6, the synchronization failure ratios for DMR with GPU-STM are considerably higher than those with TLLL. This is attributed to the substantially larger read and write sets in each transaction with GPU-STM in DMR, which increases the probability of synchronization conflicts.

SBank. As shown in Table 6, the main warp stalling reason for SBank shifts to barrier when using GPU-STM, which extensively utilizes barrier primitives to coordinate threads within warps. This shift occurs due to SBank's tiny critical section/transaction executing time, making its performance highly susceptible to synchronization.

TPCC. TPCC yields several interesting results. Firstly, as shown in Figure 6, the synchronization failure ratio is lower in TLLL compared to GPU-STM on both GPUs. This is attributed to the complicated critical sections/transactions in TPCC, which involve numerous b+ tree operations. Given the inherent challenges in parallelizing b+ tree insertions, GPU-STM employs atomic-based locking to ensure the correctness of b+ tree operations, just like the DrTM [42] does. While this boosts throughput, it leads to a higher synchronization failure (the failures of atomic based locking and transaction).

Besides, TLLL_R shows a higher synchronization failure ratio than TLLL_T for TPCC-1 and TPCC-2. However, unlike many other GPUSync applications, it exhibits poorer performance due to the costs of increased synchronization failures. Furthermore, TLLL_R's performance is even worse than GPU-STM_R for TPCC-2, further suggesting the significant synchronization failure overhead incurred by TLLL on the RTX 3080.

KM. Table 6 shows that for TLLL, the avg_acthreads_p-erwarp for KM-3 and KM-4 on the TITAN X nearly reaches 32, indicating a minimal frequency of branch divergences. Despite having a lower avg_ewarps_percycle compared to TLLL_R, they introduce a higher synchronization failure ratio and yield better performance. Besides, similar to DMR, GPU-STM incurs a higher synchronization failure ratio than TLLL_R due to the large read and

write sets involved in each transaction.

LB. Unlike other applications in GPUSync, LB concurrently accesses the shared data across TBs. In addition, it frequently utilizes the TB-level barrier primitive (i.e., __syncthreads()) to coordinate between TBs. As a result, as shown in Table 6, for LB-1 and LB-2, which have small working sets, barrier emerges as the main stalling reason. Notably, without the data sharing across individual threads in LB, the avg_acthreads_perwarp with TLLL and GPU-STM exhibit similar patterns across the two GPUs. Moreover, due to the large read and write set for shared data per transaction in GPU-STM, it incurs a higher synchronization failure ratio than TLLL.

GN. As shown in Table 6, with the small working set of kernel ① and ② in GN, the main warp stalling reason shifts to wait on both TLLL and GPU-STM. This shift indicates a highly optimized kernel in Nsight Compute. Moreover, GPU-STM outperforms TLLL for kernel ① of GN, as it efficiently aborts transactions upon detecting non-unique segments, thereby reducing the number of transactions to be executed and enhancing overall performance. Lastly, for kernel ② of GN, TLLL_T exhibits a higher synchronization failure ratio compared to TLLL_R.

HT. As shown in Figure 6, TLLL_R shows poorer performance compared to TLLL_T and GPU-STM_R for HT-1. Furthermore, as highlighted in Table 6, HT-1 with TLLL_R displays a significantly higher avg_acthreads_perwarp and slightly smaller avg_ewarps_percycle than TLLL_T and GPU-STM_R, indicating an increased actual parallelism. However, similar to TPCC-2, this increased parallelism drags down performance, as the costs of increased synchronization failures exceed the gains of increased parallelism. Besides, as shown in Table 6, the main warp stalling reason for HT-4 with GPU-STM shifts to barrier.

RA. As shown in Table 6, RA-4 with TLLL_T exhibits a markedly low avg_ewarps_percycle, resulting in suboptimal performance that is even poorer than GPU-STM_T. Furthermore, with the tiny critical section/transaction length, RA-4 with GPU-STM shifts the main warp stalling reason to barrier. In addition, as shown in Table 2, RA-1 and RA-3 are set up for 100% write-only transactions, indicating all branch divergences are induced by the synchronization. Thus, the avg_acthreads_perwarp can expose the branch divergences caused by synchronization operations and failures.

6.3 Insights

Based on the above evaluation results, we derive the following insights.

Firstly, independent thread scheduling can introduce indeterminate errors for fine-grained synchronization mechanisms that rely on the SIMT execution paradigm. As illustrated in Subsection 5.1, these errors are highly complex to avoid with the current hardware support and available programming primitives. Moreover, independent thread scheduling leads to poorer performance for certain GPUSync applications. Examples include KM-3 and KM-4 with TLLL. Despite the RTX 3080's higher clock rate and larger L2 cache compared to the TITAN X, they show a performance slowdown of 23% and 21%, respectively. Therefore, it is imperative to develop more convenient warp-level convergence mechanisms and to optimize warp-level reconverging strategies on modern GPUs with independent thread scheduling.

Secondly, GPUs' vast computing units enable significant parallelism, which constitutes a double-edged sword for applications with fine-grained synchronization. As observed in many applications in GPUSync, increased parallelism boosts throughput but simultaneously escalates the synchronization failure ratio (e.g., DMR-2, DMR-3, DMR-4, LB and kernel ① in GN with TLLL). The additional synchronization failure overhead, as demonstrated in TPCC-1, TPCC-2, and HT-1 with TLLL_R, can negate the gains from high parallelism, ultimately impeding performance. To be noticed, this scenario is far more common on GPUs than on CPUs due to the limited parallelism of CPUs. Therefore, it is especially critical on GPUs to explore strategies for identifying optimal parallelism levels or controlling concurrency for fine-grained synchronization applications.

Thirdly, waiting for memory accesses is the main stalling reason for many applications in GPUSync, such as DMR, TPCC, and KM. Therefore, reducing memory access latency is crucial for improving the performance of applications with fine-grained synchronization on GPUs. Currently, most fine-grained synchronization methods store metadata in global memory and heavily rely on atomic operations. For instance, in workloads like SBank-3, HT, and RA with TLLL, atomic instructions constitute over 20% of all memory access instructions (shown in Table A1 in Appendix A). Due to the long latency of atomic operations to global memory, this severely degrades performance. Furthermore, the results in Appendix A reveal a high degree of uncoalesced memory access, further exacerbating memory access latency. Therefore, significant potential for performance improvement exists in GPU applications with fine-grained synchronization, which can be achieved by minimizing atomic operations to global memory in synchronization mechanisms and enhancing memory access coalescing.

Lastly, it is imperative for researchers to mitigate branch divergences for applications with fine-grained synchronization. As the avg_acthreads_perwarp values indicate, for most GPUSync applications on both GPUs, more than

half of the threads (i.e., 16) within a warp remain inactive during execution (e.g., DMR-1, DMR-2, DMR-4, HT-2, HT-3, RA-1, et al.). This indicates frequent branch divergences, which lead to poor performance and low hardware utilization.

7 Conclusion

GPUSync is a GPU benchmark suite that enables systematic exploration of architectural support for synchronization and provides a standardized platform for the evaluation of fine-grained synchronization. It adequately covers a wide range of application domains and stresses most cases of fine-grained synchronization scenarios. To facilitate the evaluation on GPUs, we categorize the diverse synchronization behaviors into four key features, and further design comprehensive evaluation parameters for the eight GPUSync applications. Additionally, we run GPUSync with a classical locking and TM system across an old and a modern GPU, respectively, and present a thorough evaluation methodology by carefully filtering out the metrics necessary for assessing the performance of applications with fine-grained synchronization.

By providing applications with a wide range of synchronization characteristics and conducting a comprehensive evaluation for GPUSync, we identify four pivotal insights on both the architectural support and software development for fine-grained synchronization. These insights reveal the impact of GPU specific factors—such as independent thread scheduling, massive parallelism, memory system constraints, and branch divergence—on the correctness and performance of fine-grained synchronization. We believe these insights can motivate further research on fine-grained synchronization on GPUs.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2023YFB3002003) and National Natural Science Foundation of China (Grant Nos. 62202317, 92373110, U22A2028, 62341411, U23B2020).

References

1 Gao L, Xu Y, Wang R, et al. Thread-level locking for SIMT architectures. IEEE Trans Parallel Distrib Syst, 2020, 31: 1121–1136
2 Wang K, Fussell D, Lin C. Fast fine-grained global synchronization on GPUs. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 793–806
3 ElTantawy A, Aamodt T M. Warp scheduling for fine-grained synchronization. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture, 2018. 375–388
4 Gao L, Wang J, Zhang W. Adaptive contention management for fine-grained synchronization on commodity GPUs. ACM Trans Archit Code Optim, 2022, 19: 1–21
5 ElTantawy A, Aamodt T M. MIMD synchronization on SIMT architectures. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016. 1–14
6 Jacob N, dePaul M, Palmieri R. Don't forget about synchronization! Guidelines for using locks on graphics processing units. Concurr Comput Pract Exper, 2020, 34: 1–18
7 Li A, van den Braak G-J, Corporaal H, et al. Fine-grained synchronizations and dataflow programming on GPUs. In: Proceedings of the 29th ACM International Conference on Supercomputing, 2015. 109–118
8 Chen S, Peng L. Efficient GPU hardware transactional memory through early conflict resolution. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture, 2016. 274–284
9 Chen S, Peng L, Irving S. Accelerating GPU hardware transactional memory with snapshot isolation. SIGARCH Comput Archit News, 2017, 45: 282–294
10 Fung W W L, Aamodt T M. Energy efficient GPU transactional memory via space-time optimizations. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013. 408–420
11 Fung W W L, Singh I, Brownsword A, et al. Hardware transactional memory for GPU architectures. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011. 296–307
12 Ren X, Lis M. High-performance GPU transactional memory via eager conflict detection. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture, 2018. 235–246
13 Xu Y, Wang R, Goswami N, et al. Software transactional memory for GPU architectures. In: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, 2014. 1–10
14 Villegas A, Asenjo R, Navarro A, et al. Lightweight hardware transactional memory for GPU scratchpad memory. IEEE Trans Comput, 2018, 67: 816–829
15 DePaul M, Jacob N, Ahmed H, et al. KVCG: a heterogeneous key-value store for skewed workloads. In: Proceedings of the 14th ACM International Systems and Storage Conference, 2021. 1–12
16 NVIDIA. Volta. 2017. https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
17 NVIDIA. NVIDIA CUDA SDK Code Samples. 2020. https://developer.nvidia.com/cuda-code-samples
18 Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2009. 44–54
19 Nasre R, Burtscher M, Pingali K. Morph algorithms on GPUs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013. 147–156
20 Stratton J A, Rodrigues C, Sung I-J, et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, 2012. http://impact.crhc.illinois.edu/Shared/Report/impact-12-01.parboil.pdf
21 O'Neil M A, Burtscher M. Microarchitectural performance characterization of irregular GPU kernels. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2014. 130–139
22 Karimov J, Rabl T, Markl V. Polybench: the first benchmark for polystores. In: Proceedings of the 10th TPC Technology Conference, 2018. 24–41
23 Danalis A, Marin G, McCurdy C, et al. The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units, 2010. 63–74
24 Naderan-Tahan M, Eeckhout L. Cactus: top-down GPU-compute benchmarking using real-life applications. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2021. 176–188

25 Gómez-Luna J, Hajj I E, Chang L-W, et al. Chai: collaborative heterogeneous applications for integrated-architectures. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2017. 43–54
26 Sun Y, Gong X, Ziabari A K, et al. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2016. 1–10
27 Narang S, Diamos G. DeepBench. 2016. https://svail.github.io/DeepBench
28 Mattson P, Reddi V J, Cheng C, et al. MLPerf: an industry standard benchmark suite for machine learning performance. IEEE Micro, 2020, 40: 8–16
29 Karki A, Palangotu K C, Mysore S S, et al. Tango: a deep neural network benchmark suite for various accelerators. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2019. 137–138
30 Zhu H, Akrout M, Zheng B, et al. Benchmarking and analyzing deep neural network training. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2018. 88–100
31 Hu B, Rossbach C J. Altis: modernizing GPGPU benchmarks. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2020. 1–11
32 Sinclair M D, Alsop J, Adve S V. HeteroSync: a benchmark suite for fine-grained synchronization on tightly coupled GPUs. In: Proceedings of the International Symposium on Workload Characterization, 2017. 239–249
33 Che S, Beckmann B M, Reinhardt S K, et al. Pannotia: understanding irregular GPGPU graph applications. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2013. 185–195
34 Gramoli V. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015. 1–10
35 Ansari M, Kotselidis C, Watson I, et al. LEE-TM: a non-trivial benchmark suite for transactional memory. In: Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing, 2008. 196–207
36 Guerraoui R, Kapalka M, Vitek J. Stmbench7: a benchmark for software transactional memory. ACM SIGOPS Oper Syst Rev, 2007, 41: 315–324
37 Minh C C, Chung J, Kozyrakis C, et al. STAMP: Stanford transactional applications for multi-processing. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2008. 35–46
38 Kestor G, Karakostas V, Unsal O S, et al. RMS-TM: a comprehensive benchmark suite for transactional memory systems. In: Proceedings of the International Conference on Performance Engineering, 2011. 335–346
39 Hong S, Oguntebi T, Casper J, et al. Eigenbench: a simple exploration tool for orthogonal TM characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2010. 1–11
40 Cheng S-W, Dey T K, Shewchuk J, et al. Delaunay Mesh Generation. Boca Raton: CRC Press, 2013
41 Qian S, Goel A. Massively parallel multi-versioned transaction processing. In: Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation, 2024. 765–781
42 Chen H, Chen R, Wei X, et al. Fast in-memory transaction processing using RDMA and HTM. ACM Trans Comput Syst, 2017, 35: 1–37
43 Villegas A, Asenjo R, Navarro A, et al. Hardware support for scratchpad memory transactions on GPU architectures. In: Proceedings of the European Conference on Parallel Processing, 2017. 273–286
44 Ashkiani S, Martin F-C, John D. A dynamic hash table for the GPU. In: Proceedings of the International Parallel and Distributed Processing, 2018. 419–429
45 The H-Store Team. SmallBank Benchmark. 2018. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank
46 Olszewski M, Cutler J, Steffan J G. JudoSTM: a dynamic binary-rewriting approach to software transactional memory. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, 2007. 365–375

Appendix A Breakdown of memory subsystem behavior

To further identify the specific factors that contribute to memory access being the main warp stalling reason, we conduct a detailed evaluation of the memory subsystem. This analysis is performed using the NVIDIA Nsight Compute profiling tool on the RTX 3080 GPU, focusing on applications where memory access is identified as the main warp stalling reason. For these applications, we collect detailed hardware performance counters for both the TLLL and GPU-STM implementations, to enable a comparative investigation into the root causes of memory latency.

Specifically, we collect the following key metrics to quantify memory subsystem behavior: (1) the percentage of atomic instructions to global memory among all memory access instructions (P_Atomic in Table A1), which helps identify the contribution of atomic instructions and isolates the performance impact of synchronization primitives; (2) L2 cache throughput (T_L2 in Table A1), defined as a percentage of the peak sustained rate achieved during elapsed cycles, indicating whether the L2 cache is a saturation point; (3) memory throughput (T_Mem in Table A1), measured similarly as a percentage of peak sustained rate, to assess potential DRAM bandwidth saturation; and (4) the exceeded sector access ratio (R_Sector in Table A1), caculated as the percentage of memory requests that fetch excess sectors than theoretically required due to uncoalesced or misaligned accesses, serving as a direct indicator of memory access inefficiency.

As shown in Table A1, the measured L2 and memory throughput values are generally low, with the highest recorded being 14.08% for RA-3 with TLLL and 17.77% for HT-1 with GPU-STM. Such a low percentage of the peak sustained rate suggests that the memory subsystems are not bandwidth-bound, which is unlikely to be the primary factor that incurs long memory access latency.

Additionally, the results reveal two critical issues. Firstly, a significant source of memory pressure stems from atomic operations. For several applications, such as SBank-3, HT, and RA with TLLL, and TPCC-1 and HT-1 with GPU-STM, atomic operations constitute over 20% of all memory accesses. Given the inherently long latency of atomic operations and the observed long memory accesses, we conclude that the latency of the atomic operations themselves limits the performance rather than a lack of available memory bandwidth. This finding underscores the importance of reducing atomic operation latency. Beyond architectural improvements, a promising software optimization is for developers to migrate synchronization metadata (e.g., locks) to on-chip memories, such as shared memory, to mitigate this latency.

Second, the high exceeded L2 sector access ratios across many applications point to inefficient memory access patterns characterized by poor coalescing and misalignment. Applications with fine-grained synchronization often perform random memory accesses that cannot be efficiently coalesced by the GPU's memory controllers. This inefficiency exacerbates access latency. Therefore, enhancing memory access coalescing through data structure or algorithm redesign presents a significant opportunity for performance improvement for applications with fine-grained synchronization.

Table A1 Key memory system metrics collected from the NVIDIA Nsight Compute tool.

	P_Atomic (%)		T_L2 (%)		T_Mem (%)		R_Sector (%)	
	TLLL_R	GPU-STM_R	TLLL_R	GPU-STM_R	TLLL_R	GPU-STM_R	TLLL_R	GPU-STM_R
DMR-1	2.43	6.12	6.55	4.30	7.82	4.52	58	36
DMR-2	7.00	5.00	4.33	2.60	4.37	3.10	63	38
DMR-3	2.27	5.54	8.43	3.86	10.93	4.07	63	36
SBank-3	22.05	10.31	5.11	2.76	5.46	2.76	75	29
TPCC-1	4.77	21.61	5.37	6.17	5.37	6.45	48	38
TPCC-2	1.53	11.24	3.13	6.48	3.77	6.48	41	52
KM-1	0.14	8.60	0.21	0.88	1.76	1.23	60	36
KM-2	0.10	8.25	0.27	0.92	2.42	1.51	62	35
KM-3	0.01	1.97	0.08	0.71	6.40	6.15	81	82
KM-4	0.01	2.19	0.08	0.49	10.70	7.73	82	82
HT-1	90.66	22.91	3.93	11.47	9.70	17.77	13	5
HT-2	21.95	7.21	5.45	4.82	5.45	6.95	65	29
HT-3	37.60	3.19	6.19	3.13	6.19	5.31	75	30
RA-1	42.05	10.33	9.28	6.74	9.38	9.65	73	32
RA-2	42.58	8.65	8.18	6.21	9.24	7.92	73	34
RA-3	34.72	9.10	14.08	5.91	14.08	7.58	82	32