LTPG: Large-Batch Transaction Processing on GPUs with Deterministic Concurrency Control

Jianpeng Wei[1], Yu Gu[1]*, Tianyi Li[2], Jianzhong Qi[3], Chuanwen Li[1],
Yanfeng Zhang[1], Christian S. Jensen[2], Ge Yu[1]
[1]Northeastern University, Shenyang, China, 2310745@stu.neu.edu.cn,
{guyu, lichuanwen, zhangyf, yuge}@mail.neu.edu.cn
[2]Aalborg University, Aalborg, Denmark, {tianyi, csj}@cs.aau.dk
[3]The University of Melbourne, Parkville, Australia, jianzhong.qi@unimelb.edu.au

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00296

***Abstract*—GPUs are being applied widely to batch workloads that benefit from the parallel processing capabilities of GPUs. To enable the processing of concurrent batch-based transactions on GPUs, existing systems build dependency graphs during a pre-execution phase to manage read and write operations. However, as dependency-graph maintenance introduces a substantial overhead, there is a need for more efficient transaction support to exploit the power of GPUs more fully for transaction processing. This paper proposes LTPG, a novel GPU-enabled database system that offers increased versatility and efficiency by eliminating the need for predefined read/write-sets. LTPG employs deterministic optimistic concurrency control to ensure correct transaction execution, thus avoiding the maintenance of dependency graphs. The proposed concurrency control simplifies transaction processing workflows and avoids the overhead associated with managing dependency graphs, thus resulting in improved efficiency. LTPG divides a workflow into three stages: execution, conflict detection, and write-back, leveraging the parallelism of GPUs. Moreover, several additional optimization strategies are adopted to improve system performance. Experiments with real-world workloads from two benchmarks verify LTPG can achieve effective improvement in the throughput and latency compared to the leading baselines.**

***Index Terms*—Database, deterministic concurrency control, transaction processing, GPU**

I. Introduction

Advancements in General Purpose Graphics Processing Units (GPGPUs) have opened up new possibilities for accelerating DataBase Management Systems (DBMS). Previous studies have focused on utilizing GPUs for OnLine *Analytical* Processing (OLAP) workloads and leveraging the vectorized execution model of GPUs [5], [6], [12], [32]. In contrast, little attention has been given to means of exploring GPUs to enhance the performance of OnLine *Transaction* Processing (OLTP) performance. Compared to OLAP tasks, the run-time of OLTP is typically short [4]. This means the benefit of GPU parallelism to a single transaction may be marginal. However, organizing transactions into batches holds immense potential, as the vectorized processing capabilities of GPUs enable the processing of large numbers of such transaction batches in parallel. Existing studies [4], [14] have already demonstrated the feasibility of GPU-powered OLTP systems as cost-effective alternatives to expensive multi-core CPU-powered systems. These GPU-powered systems can even achieve up to a 10-fold improvement over the benchmark multi-core database, DBx1000 [4].

When processing transactions on GPUs, existing approaches [4], [14] employ both transaction batching and deterministic pessimistic concurrency control. Batching takes advantage of the concurrent computing capabilities of GPUs, while deterministic pessimistic concurrency control ensures the correct execution of transactions within a batch.

There are two main challenges with existing systems. First, the deterministic pessimistic concurrency control requires a preprocessing phase before the actual transaction processing to obtain the read-write sets of transaction batches. This reduces system generality and increases time consumption. Second, deterministic pessimistic concurrency control causes serialization in transaction execution, which further increases transaction processing time. The difficulty in addressing these two challenges lies in the fact that they are inherent problems of the existing approaches and can only be partially mitigated by adopting certain strategies. Further, the existing solutions based on CPU systems cannot be applied directly to address the challenges we face. Due to the differences in hardware architectures, the existing solutions are not suitable for the Single Instruction Multiple Threads (SIMT) [31] architecture of GPUs. Instead, we need to design completely new solutions that are compatible with GPUs.

We propose LTPG, which is a system that utilizes deterministic optimistic concurrency control and leverages the data parallelism of GPUs for efficient transaction processing. Regarding the first challenge, LTPG can process transactions directly without pre-processing. This allows LTPG to handle a wider range of business scenarios compared to existing GPU transaction processing systems [4], [14]. Regarding the second challenge, LTPG significantly reduces the execution time of a single batch of transactions by employing transaction abort and re-execution in the next batch to resolve conflicts. This decomposes transactions into modularized reusable components. Thus, LTPG meets the same instruction requirements across multiple warps. Each warp focuses on a specific type of transaction operation such as *Select* or *Update*. Then conflicting operations can be identified through our *conflict detection* mechanism before entering the *write-back* phase. LTPG main-

* Corresponding author

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00296

tains correctness by guaranteeing serializable transactions. By organizing fine-grained transactions into mutually exclusive types, the system enables more effective management of shared resources, such as device memory. In conclusion, LTPG is built to fully utilize GPUs and accelerate the overall process.

To summarize, we make the following contributions:

- We propose LTPG, a GPU-based batch transaction processing system designed for deterministic databases. Unlike existing GPU-based systems, LTPG eliminates the need for transaction dependency graphs, resulting in reduced processing overhead and increased efficiency.
- We provide new GPU-based batch transaction processing optimizations: (1) a transaction decomposition and warp grouping for GPU data parallelism and (2) a dynamic hash bucket technique for faster conflict detection.
- We propose an optimization technique for large-batch-based transaction processing, which encompasses three sub-techniques that reduce transaction aborts in high-contention scenarios.
- We conduct experiments using the TPC-C [1] and YCSB [7] benchmarks. The results show that LTPG significantly improves transaction throughput by up to $1.9\times$ over the state-of-the-art GPU-based OLTP systems and up to $7\times$ over the state-of-the-art multi-core CPU OLTP systems.

The rest of the paper is organized as follows. In Section III and Section III, we discuss the background and related work. We give an overview of our proposed system in Section IV. In Section V, we detail LTPG and our optimization techniques. We cover our empirical study and its findings in Section VI. Finally, we conclude the paper with Section VII.

II. Background

A. Introduction to GPU programming

We briefly introduce four important concepts in GPU programming. (i) A **warp** is a batch of threads that perform instructions simultaneously, ensuring efficient and simultaneous processing. (ii) **Atomic operations** carry out read-modify-write tasks on shared data in a single step. They prevent errors that occur when multiple threads try to modify the same data simultaneously. (iii) **Synchronization** in computing, particularly in the context of parallel processing, refers to the coordination of concurrent processes or threads. It ensures the correct execution and data consistency of parallel processes. Existing works [19], [20] demonstrate that GPUs have significant parallel advantages in accelerating computations.

B. Main memory transaction systems

Main memory transaction systems process transactions without relying on disk-based storage, providing high performance due to fast data access [8], [15], [17], [21], [22], [24], [26], [34]–[36], [38]. Some of the systems designed for highly concurrent workloads [35], [38], [18], [22] adapt their concurrency control protocols to support highly concurrent workloads. DBx1000 [37] is a database focused on OLTP performance on multi-core CPUs. Bamboo [13] explores a transaction concurrency control protocol that uses an improved two-phase locking mechanism on multi-core databases, with a primary focus on transactional performance on popular data. The main-memory Silo database [34], [39] introduces an Optimistic Concurrency Control (OCC) protocol, which allows for a significant reduction in contention and contention-related overhead in highly concurrent transaction processing environments. However, Silo [34] suffers from performance crashes under high contention. TicToc [38] aims to maintain the advantages of OCC at low contention and mitigates its shortcomings at high contention. Specifically, it employs two timestamps: the write timestamp is similar to the timestamp of OCC records, whereas the read timestamp allows TicToc [38] to commit some conflicting transactions by reordering them. Timestamps become more expensive to maintain than OCC, but reordering has benefits for highly competitive workloads. Multi-Version Concurrency Control (MVCC) systems [18], [22], [2], [27] keep multiple versions for each record. These versions allow more transactions to be committed by reordering. As for high-contention transaction optimization, timestamp splitting splits records based on workload characteristics to optimize I/O. It has a coarser granularity, which reduces the overhead of fine-grained locking and is sufficient to reduce conflicts. In general, *Deterministic Concurrency Control* is often used to reduce the replication overhead and communication overhead of scalable transaction processing [16], [25]. The core idea of deterministic concurrency control is to ensure that different replicas always produce the same result independently (as long as the same input transaction is given) to avoid the use of expensive commit and replication protocols [11], [28]–[30].

C. Deterministic Databases Transaction Systems

A deterministic database transaction system ensures predictable and repeatable transaction outcomes, promoting data consistency and stability. Existing studies can be categorized into CPU-based systems [10] [9] [33] [23] and GPU-based systems [14] [3] [4].

CPU-based systems. BOHM [10] employs a two-step process for transactions. In the first step, it inserts primary key placeholders of the write set into a multi-version storage layer along with the Transaction ID (TID). In the second step, each transaction reads a specific version for each key in its read set (the one with the largest TID up to the current TID) to ensure determinism. PWV [9] decomposes each input transaction into fragments, allowing each fragment to access a non-overlapping partition of a database table. A dependency graph is then computed to meet data dependency and commit dependency requirements. Calvin [33] utilizes a single-threaded lock manager to grant read/write locks based on pre-declared read/write-sets. After acquiring all locks for a transaction, the lock manager assigns it to an available worker thread for execution.

Aria [23] executes transactions in two steps. In the first step, each transaction reads from the current database snapshot and saves writes in a local write set. It adopts a single-version approach, buffering write transactions until the end

TABLE I
SUMMARY OF DIFFERENCES BETWEEN SCHEMES

System	Versioning	Initialization	Execution	Granularity[a]	Need R/W set?	Handel skew?	Handle contention?	Parallelism[b] mode
BOHM [10]	Multi	Partitioned	Shared	1 thread	Yes	No	No	task
PWV [9]	Single	Partitioned	Partitioned	several threads	Yes	No	Yes	task
Calvin [33]	Single	1 thread	Shared	1 thread	Yes	Yes	No	task
Aria [23]	Single	No need	Shared	1 thread	No	Yes	Yes	task
Bamboo [13]	Single	No need	Shared	1 thread	No	Yes	Yes	task
GPUTx [14]	Single	Using GPU	Shared	1 thread	Yes	No	No	task
GaccO [4]	Single[c]	Using GPU	Partitioned	several threads	Yes	No	No	task
LTPG	Single	Using CPU	Shared	several warps	No	Yes	Yes	data

[a] Transaction execution granularity.
[b] Task parallelism or data parallelism.
[c] GaccO has a single version in GPU but multiple versions in CPU.

of the batch. Once all transactions in the batch are executed on the replicas, they proceed to the second phase, i.e., the commit phase. This phase involves multiple independent worker threads on the replica. If a transaction conflicts with an earlier one (smaller TID) due to operations performed, it is aborted and rescheduled for the next batch. TIDs are assigned based on transaction arrival time. Non-aborted transactions are committed by the system.

GPU-based systems. GPUTx [14] employs a batch transaction model where transactions are predefined and set up as stored procedures, executed as CUDA device functions for optimal parallelism on the GPU at the data field level. To guarantee correct execution, it uses a T-dependency graph—a directed acyclic graph that maps out data dependencies to maintain the integrity of batch executions.

GalOP [3] and GaccO [4] are the same systems using a deterministic concurrency scheme to execute batch transactions on the GPU while the CPU executes non-batch transactions. The system optimizes data movement overhead by storing primary copies of all tables on the CPU and only retaining secondary copies of tables needed for GPU transactions in GPU memory. To maintain data consistency, updates are propagated between the CPU and GPU copies of the tables.

We give more details about GaccO [4] below. GaccO [4] utilizes pre-processing on the GPU to determine the conflict order of transactions within a batch. This conflict order is then employed during transaction execution to manage concurrent access to shared resources. During the pre-processing stage, GaccO [4] creates access tables and other auxiliary tables. These tables facilitate efficient execution of the conflict order determined by GaccO [4]. To achieve this, GaccO [4] must first arrange its access table in ascending order based on TIDs, enabling it to identify the appropriate sequence of transactions capable of accessing contested tuples. To guarantee correct operation, GaccO [4] employs supplemental scheduling techniques to prioritize the initial transaction appearing in the overall order for processing by the GPU's internal scheduler. Additionally, GaccO [4] offers two innovative enhancements. Firstly, it converts locks and mutexes applied to objects into interchangeable atomic actions performed on equivalent pieces of information. Secondly, GaccO [4] leverages intra-transaction parallelism instead of sequential steps within each transaction, thereby boosting performance.

III. RELATED WORK

Limitations of CPU-based systems. BOHM [10], PWV [9], and Calvin [33] face limitations due to explicit and implicit dependency graphs, as shown in Fig. 1. Aria [23], in contrast, does not rely on dependency graphs. However, it is less efficient when handling significantly longer transactions in a batch due to its thread assignment approach. Furthermore, Aria [23] is not suitable for GPUs as it struggles with data conflicts arising from a larger number of threads, especially in large transaction batches. DBx1000 [37] and Bamboo [13] are databases designed for multi-core CPUs. Many of their optimizations are not suitable for GPU's Single Instruction Multiple Thread (SIMT) architecture.

Limitations of GPU-based systems. Both GPUTx [14] and GaccO [4] suffer from inefficiency when executing highly contended transactions serially, due to leveraging dependency graphs, as shown in Fig. 1. In particular, GaccO [4] incurs data transmission costs between CPU and GPU. For large transaction batches, the dependency graph complexity and branching increase, resulting in inefficiency during transaction processing acceleration.

Comparison between LTPG and existing GPU-based systems. As the biggest difference between LTPG and existing GPU-based systems, LTPG does not require a pre-execution phase to obtain the read and write sets. In addition, LTPG allows multiple transactions to be batched and executed simultaneously on the GPU, ensuring high performance. In contrast, GaccO requires transactions to be classified and can only process the same type of transactions within a batch. When facing a workload with mixed transaction types, GaccO may be difficult to generalize.

Table I summarizes the differences between existing systems and LTPG. Note that, LTPG eliminates the need to construct dependency graphs for deterministic transaction concurrency. At the same time, it transforms task parallelism into data parallelism, making transactions more adaptable to GPU execution architectures. In addition, it reduces the data transfer overhead by residing the data in GPU memory and achieves improvement in GPU utilization.

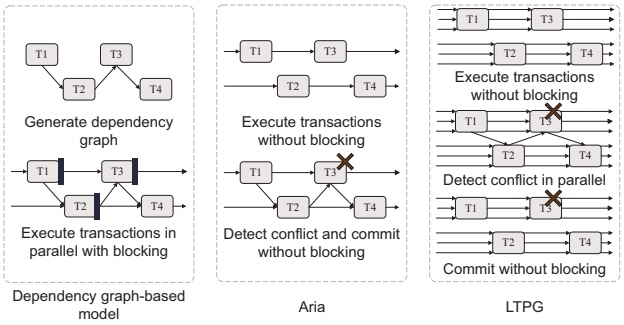


Fig. 1. Comparison with existing models.

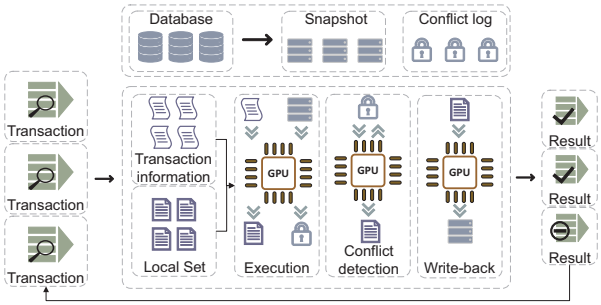


Fig. 2. Execution flow of LTPG.

IV. System Overview

Here, we introduce LTPG, a novel system that accelerates transaction processing using GPUs. In Fig. 2, LTPG involves three main steps: execution, conflict detection, and write-back. Before formally starting the execution of transactions, LTPG captures a snapshot of the current database and incoming transactions. Separate GPU memory spaces are created for local read-write sets and state sets. Next, LTPG assigns TIDs to incoming transactions based on their arrival order. Finally, the snapshots, transactions, and related data structures are transmitted to the GPU for execution.

Execution. During execution, all operations are conducted using the local read and write sets, thus avoiding data updates before write-back. Additionally, each transaction must register its TIDs in a hash table linked to the relevant data items, to facilitate future conflict detection.

Conflict detection. During conflict detection, LTPG examines the TIDs stored in the hash table to identify any conflicting WAW, RAW, or WAR operations. If such conflicts exist, they are marked before proceeding to commitment.

Write-back. Here, it is decided whether to commit or abort each transaction based on the results from the conflict detection phase. Transactions that are committed write their modifications back to the database snapshot, whereas aborted transactions await re-execution or return.

Following the write-back phase, both the transactions and associated data structures are returned to the CPU, allowing aborted transactions to be rescheduled for re-execution in the next batch. Database snapshots are saved regularly to the hard drive for permanent storage. LTPG provides two data synchronization methods. The first relies on a user-defined interval for transferring data from the GPU to the CPU, streamlining the synchronization process during GPU transaction execution. Although this method involves data movement between the CPU and GPU, it simplifies data synchronization. The second only transfers the transaction read/write-set and conflict flag table to the CPU after processing each batch. This method reduces the data transfer volume significantly and avoids the need for comprehensive database traversals during synchronization, which boosts efficiency. Further, the more frequent update intervals ensure stronger consistency between CPU and GPU data, enhancing system robustness. We report the overhead of copying transaction read/write-sets in Section.VI-C.

Note that LTPG stores both database snapshots that are used in transaction processing and conflict log tables in GPU memory. This is to avoid data transfer overhead. The CPU also records each batch of transactions on the hard drive as logs. LTPG guarantees consistent transaction outcomes by assigning a unique TID to each transaction in a batch, logging it for reference. If re-execution is necessary, the system pulls the transactions from the log, while preserving their original TIDs and dependencies. This consistency in transaction handling, coupled with the same commit policy, ensures uniform commit results, ensuring LTPG's determinism.

In LTPG, it is necessary to ensure that each stage is fully completed before moving on to the next stage. However, grid synchronization alone is insufficient to guarantee the complete execution of all threads on the device. Thus, we employ the *cudaDeviceSynchronize()* primitive to achieve isolation between kernel functions. Further, if there are data conflicts for some operations within a transaction, they need to be present in the same device function. During the execution phase, data from the database snapshot is copied to local sets to prevent interference between the execution and write-back phases.

Novelty. LTPG stands out in three main ways. First, LTPG is designed as a transaction processing system tailored for the Single Instruction Multiple Thread (SIMT) architecture of GPUs and data access patterns. Its essential feature is to reorganize sub-transactions to make the most out of the capabilities of GPUs. Second, LTPG can process transactions directly without pre-processing. This allows LTPG to handle a wider range of business scenarios compared to existing GPU transaction processing systems [4], [14]. Finally, unlike other systems in its category [4], [14], LTPG does not need to maintain dependency graphs, which enables it to utilize better the concurrency offered by GPUs. LTPG also does not require pre-execution to obtain transaction read/write-sets, allowing it to adapt to more scenarios.

Comparison between LTPG and Aria. LTPG operates on in-memory databases using GPUs, while Aria is designed for distributed in-memory databases on CPUs. This leads to three main differences. First, the architectures of LTPG and Aria differ. LTPG, designed for transactional concurrency, ensures smooth GPU operation by dividing transaction processing into

Algorithm 1: Core architecture of LTPG

Input : Transactions T, Snapshot S, Conflict logs CL
Iutput: Transaction results Txr

```
1  launchKernel(execute, T, S, CL)
2  cudaDeviceSynchronize()
3  launchKernel(conflict_d, T, S, CL)
4  cudaDeviceSynchronize()
5  launchKernel(writeback, T, S, CL)
6  cudaDeviceSynchronize()
7  Function execute(T, S, CL):
8      Warp.get(instructions)
9      foreach thread in Warp do:
10         thread.exec(instructions)
11         // readMem or recordTID or recordLS
12 Function conflict_d(T, S, CL):
13     if Warp check read operations
14         foreach thread in Warp do rcheck(T, CL)
15     else
16         foreach thread in Warp do wcheck(T, CL)
17 Function writeback(T, S, CL):
18     status = getStatus(T)
19     if status == Commit
20         writeMem(T, S)
21     else
22         Abort
```

conflict detection and write-back phases. This guarantees the resolution of all conflicts before transaction commit and enables robust transaction handling by LTPG. On the other hand, Aria follows a simpler serial execution strategy, bypassing the need for a detailed conflict resolution process. Second, the two systems have different optimization foci. LTPG targets optimal GPU transaction execution, whereas Aria prioritizes ease of horizontal scaling. Finally, LTPG and Aria leverage batch processing for distinct purposes. LTPG maximizes GPU concurrency, while Aria minimizes inter-node communication overhead in distributed settings.

We highlight that porting Aria to GPUs is challenging for two main reasons. First, it requires larger batch sizes to fully utilize the numerous threads of GPUs, which unfortunately causes more transaction aborts. Second, since Aria does not sort transactions by type for each thread, directly moving it to a GPU would cause considerable thread divergence, which greatly degrades GPU performance. LTPG offers an innovative approach to optimizing GPU utilization and maintaining transactional integrity.

Algorithm.1 illustrates the flow of LTPG. Since both the database snapshot and the conflict log tables reside in GPU memory, only the transaction execution information needs to be transferred to the GPU memory before each round of transaction processing. We split all operations in a transaction into several functions. We use *readMem* to read data from GPU memory. We use *recordTID* to record TIDs in conflict logs. We use *recordLS* to keep data in local sets temporarily. We use *writeMem* to write data to snapshots in the write-back phase. We schedule GPUs in terms of warps to avoid intra-warp branch divergence. These three transaction processing phases of LTPG are implemented as separate kernel functions to ensure global synchronization (Lines 1, 3, and 5). We use CUDA synchronization instructions to ensure that the previous phase has finished before the beginning of the next phase.

V. GPU DETERMINISTIC CONCURRENCY CONTROL

This section first presents a new time cost model for measuring execution efficiency and compares it with that of the state-of-the-art GPU-based OLTP system Gacco [4]. Then it proposes various strategies in LTPG to enable LTPG's deterministic optimistic concurrency control approach for batch processing with GPU vectorized computation (see Section V-B–V-E).

A. Time Cost Analysis

Time cost model of Gacco. We provide a detailed description of the deterministic transaction processing in GaccO in Section II. The time cost of GaccO [4], denoted as T_G, is calculated as follows.

$$T_G = T_{gpp} + T_{gs} + T_{ge} + T_{gt}, \tag{1}$$

where T_{gpp} is the time cost of the preprocessing phase in GaccO [4]. We use T_{gs} to denote the time cost of sorting. We define T_{ge} as the time cost of the execution phase. We use T_{gt} to denote the time cost of data transfer.

transaction starts, and T_{dth} is the time cost of transferring updated data to the host.

Time cost model of LTPG. Based on the system described in Section IV, the time cost of LTPG, denoted as T_L, is calculated as follows.

$$T_L = T_{le} + T_{lc} + T_{lw} + T_{lt}, \tag{2}$$

where T_{le} is the time cost of the execution phase, T_{lc} is the time cost of the conflict detection phase, T_{lw} is the time cost of the write-back phase, T_{lt} is the time cost of data transfer.

Comparison between the two time cost models. The main time cost of GaccO is the time spent on transaction preprocessing and the deterministic concurrent execution of transactions. Also, LTPG needs to consider the number of aborted transactions caused by data conflicts, and LTPG indeed has a higher abort rate than Gacco [4]. This is because LTPG trades abort for better parallelism and versatility, which is its main advantage. In particular, existing systems [4], [14], which rely on dependency graphs to ensure that all transactions are processed, encounter challenges of degenerating to serialized execution when facing high-contention transactions. This issue adversely impacts the latency of all transactions in a batch.

We present a detailed analysis of the batch-level latency for LTPG and GaccO [4] in Section VI-C. Due to its superior concurrency and significantly lower latency, LTPG exhibits a much higher overall throughput than GaccO [4]. Moreover, with the high-contention transaction optimization incorporated into LTPG, robust commit performance is guaranteed, even for transactions characterized by high contention.

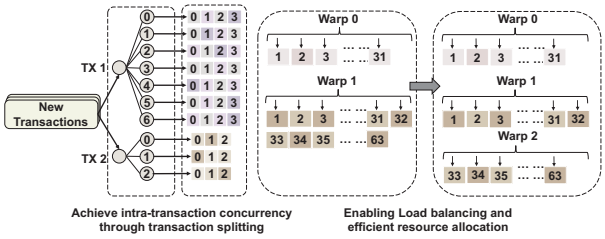


Fig. 3. Example of adaptive warps division.

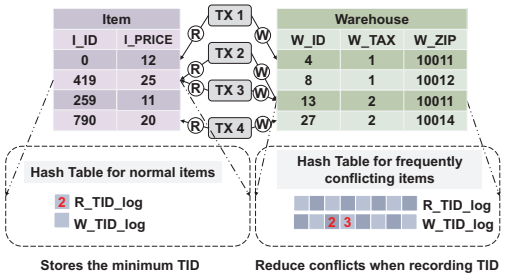


Fig. 4. Example of recording TIDs in the conflict log.

B. Adaptive Warp Division for Data Parallelism

Earlier, we mentioned that GPUs utilize Single Instruction Multiple Thread (SIMT) [31] instructions so that multiple threads perform identical instructions simultaneously. To optimize GPU processing, we separate transactions into fine-grained sub-transactions and assign each group of sub-transactions to a group of warps for concurrent execution using data parallelism. During the processing of batch transaction instances, the batch sub-transaction instances are distributed to multiple CUDA warps for execution. In LTPG, we reduce branch divergence by dividing warps and minimizing the execution of different instructions within each warp. This hardware-level approach reduces warp divergence and improves system performance. LTPG exploits adaptive warp division, by assigning collections of similar sub-transactions to worker warps from predefined types during execution, conflict checking, and commit. This strategy improves load balancing and is especially efficient for smaller transaction batches. Moreover, LTPG prevents branch divergence within warps by reconstructing batch transactions into sub-transactions and organizing them into sets. This ensures that each warp handles only one type of sub-transaction.

Example 1. *In Fig. 3, TX 1 and TX 2 are different kinds of transactions. Both are split into fine-grained sub-transactions that are processed concurrently by warps. The adaptive warp division strategy ensures optimal resource allocation, which takes advantage of the warp scheduling unit provided by the GPU architecture. In the right part of Fig. 3, the system allocates the appropriate number of warps to perform the task based on the load. We strive to keep warps fully occupied with up to 32 threads executing the same instructions. Moreover, we organize the data storage efficiently by storing related data contiguously in either global or shared memory, facilitating merged accesses and minimizing conflicts.*

C. Dynamic Hash Buckets for Conflict Detection

Considering their high query speeds, we employ hash tables for conflict recording and detection purposes. Each operation records its associated TID in the appropriate hash table element along with the modified data. After all operations have finished recording their TIDs, they later check their designated elements to determine whether any collisions occurred.

Handling memory access contention is crucial in the execution phase of LTPG since numerous threads record TIDs concurrently within batch transactions. Unlike databases like Aria [23], systems based on GPUs rely heavily on parallelism to achieve efficiency. Serial memory update delays are unacceptable because they hinder parallelism. When multiple threads attempt to access the same memory location simultaneously through atomic operations, their execution becomes sequential, negating any potential speedup offered by parallelism. As a result, collision handling mechanisms for hash tables must be established to address instances where multiple threads attempt to update the same hash table element.

To address this issue, we propose to use dynamic hash buckets and to adjust their sizes based on the frequency of data access. This approach helps alleviate the negative effects of atomic operations on concurrent accesses, thereby improving overall performance. It is important to note that the use of atomic operations themselves does not decrease the performance of LTPG. Rather, the issue arises when such operations are serialized.

LTPG enables developers to pre-mark popular data tables and also is able to identify such tables in real-time. Specifically, it evaluates the access frequency of a data table t using the formula $E = \frac{T}{D}$, where T is the number of transactions accessing t and D is the total number of rows in t. A data table is considered popular if $E > 1$.

GPU threads dedicated to processing identical data are associated with the same primary data, denoted as *key*. We employ the hash function $h(key) = key \mod s_h$ to map a GPU thread to a hash bucket, where s_h is the size of the hash table. GPU threads processing distinct data may be hashed to the same bucket, causing a collision. To resolve this, we use open addressing with the linear probing function $h(key, i) = (key + i) \mod s_h$, where $i = 0, 1, \cdots, s_h - 1$.

We denote the size of the hash bucket as s_u and implement two types of hash tables, standard-sized and large-sized. In each hash bucket of both types, we maintain two kinds of TIDs for read and write operations and store the minimum TIDs for both operations. A standard-sized hash table, with each bucket uniformly sized at $s_u = 1$, is implemented when the access frequency of the table E equals 1. If multiple GPU threads are hashed to the same standard-sized bucket, they are serialized. However, our settings for the hash bucket size and the identification of popular data ensure that this is rare.

A large-sized hash table is implemented when E exceeds 1. The size of each bucket is set to $s_u = \lceil \frac{E}{WS} \rceil \times WS$, where *WS* is the size of the corresponding warp. When multiple GPU

threads are hashed to the same large-sized bucket, we re-hash them by locating the sub-bucket within the large-sized bucket. Each GPU thread is assigned a TID after being hashed to the large-sized bucket, and we employ the hash function $h(key) = \text{TID} \mod s_u$ for re-hashing. Any remaining conflicts after rehashing result in thread serialization.

Employing a large-sized hash table offers two advantages. First, it mitigates the serialization of atomic operations. Second, it enables threads to mark their TIDs in separate slots, even when one is already in use, which reduces wait times and prevents bottlenecks. This is particularly beneficial when multiple threads access the same data rows concurrently. We provide an example that compares the use of a standard-sized hash table with that of a large-sized one.

Example 2. *Fig. 4 illustrates a scenario in which four transactions concurrently access both the Item table and the Warehouse table. When the batch size is* 2^{14} *and the number of warehouses is 32, the warehouse hash table falls into the state where* E *exceeds 1. At this time, a hash bucket of size* s_u *is arranged for recording conflicting TID information. Our hash bucket uses* atomicMin *to ensure data accuracy. Locking is not used because it is a very inefficient form of data access in CUDA. Locking is unnecessary to guarantee data accuracy with great overhead when ordinary atomic operations can be ensured. Both TX 2 and TX 3 in Fig. 4 need to be written to Warehouse 13. These two transactions have a WAW conflict. In the hash bucket, they use* $h(key) = TID \mod s_u$ *to determine the write location of the hash bucket and then record their TIDs. In the conflict detection phase, LTPG reads out all the TIDs in the hash bucket and selects the transaction with the smallest TID which is then allowed to update the row. The remaining transactions are marked as having a WAW conflict. On the left side of Fig. 4, both TX 2 and TX 3 need to read the data of Item 419. However, because the entire Item does not fall into the category of frequently accessed data items described earlier, its hash bucket has only one record bit. At this point, LTPG saves the smaller TID to the record.*

We highlight that using a large-sized hash table has a negligible impact on memory usage. This is because LTPG only enlarges hash buckets for frequently accessed data, which is typically limited in real-world scenarios. Our experiments confirm that large hash tables use minimal memory—just 0.053% to 0.055% (see Table VIII in Section VID). Despite the small size of popular data, it can lead to major performance issues. A large hash table helps avoid these bottlenecks by reducing the need to serialize atomic operations. The results are clear: large hash tables cut atomic operation latency by up to five times compared to standard tables (see Table VIII in Section VID). Overall, our hash table design greatly enhances transaction efficiency with very small memory usage.

D. Optimization for High-contention Transactions

This section considers the detailed optimization scheme for highly contentious transactions. Our strategy involves implementing a reordering technique for deterministic optimistic concurrency control, which has already proven its

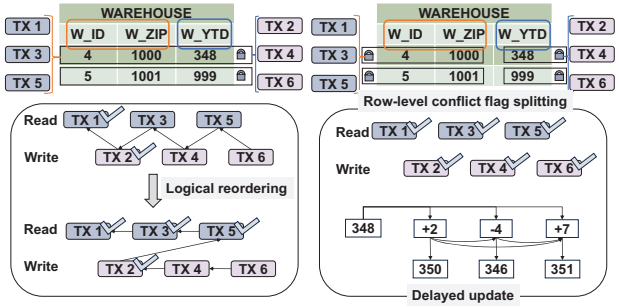


Fig. 5. Optimization of high-contention transactions.

effectiveness and accuracy according to the Aria [23], called logical reordering. By applying this method, we enable logical reordering through regular commits even when only RAW or WAR conflicts exist. As a result, the executed transactions follow a serializable execution sequence that deviates from the globally assigned monotonic timestamps, thus minimizing the risk of write skew and reducing contention.

Next, we introduce a method for splitting row-level timestamps into separate fields, called row-level conflict flag splitting. This allows us to manage conflicting operations independently by using different conflict logging tables for attributes that are updated frequently. By doing so, we ensure that operations related to specific tables and attributes are not impacted by changes made by others. For instance assume that operations involving A table in a certain type of transaction require frequent updates to attribute a, whereas the operations in other transactions do not require to read or write attribute a. With our timestamp splitting approach, we can reduce unnecessary transaction aborts resulting from overlapping write transactions affecting the same rows in the table.

Having enabled the splitting of row-level timestamps, we provide a concurrent execution strategy for frequent WAW conflicting attribute operations, called delayed update strategy. In this strategy, the execution of frequently conflicting data is delayed until the write-back phase, avoiding its participation in the conflict detection phase, thus preventing other transactions from being aborted prematurely. Specifically, the delayed update strategy is primarily used to handle the frequently updated attributes mentioned earlier. If the update operation involves addition or subtraction calculations on attributes, the prefix sum algorithm can be applied to efficiently process them in parallel on the GPU. If the update operation involves complex calculations, the efficient calculation can still be carried out on the GPU using the intra-warp communication mechanism. After execution, LTPG writes the result of the last committed operation back to the database to ensure correctness. We proceed to give an example that incorporates all the optimizations mentioned in this section.

Example 3. *Transactions Tx 1, Tx 3, and Tx 5 need to read the row with wid=4, and they need to read the value of attribute W_ZIP. Transactions Tx 2, Tx 4, and Tx 6 need to write to wid=4, and they need to modify the value of attribute W_YTD.*

Without using all the optimizations in Section V-D, only Tx 1 and Tx 2 can be committed due to dependencies between the transactions. However, when we use logical reordering optimization, the transactions that can be committed become Tx 1, Tx 2, Tx 3, and Tx 5. In this case, the logical order of these four transactions becomes Tx 1, Tx 3, Tx 5, and Tx 2, while still maintaining their original TIDs.

When we use the row-level conflict splitting strategy and delayed update strategy, all transactions can be committed. In this case, LTPG uses the same warp to process transactions that operate on the same data row, such as Tx 2, Tx 4, and Tx 6 in this example. Within the warp, each thread broadcasts its modifications to the data row to other threads and merges modifications from threads with smaller thread IDs (thID). Then the original data of the data row is read by all threads within the warp, and each thread completes its modifications to the original data. Finally, the thread with the largest thID writes its modifications back to the database to complete the entire delayed update processing.

E. Other Optimization to LTPG

We proceed to present additional optimizations that are incorporated into LTPG.

Selective memory mode adjustments. LTPG leverages selective memory mode adjustments to optimize memory allocation between zero-copy memory and unified memory according to the database size and the transaction batch size. Storing data in zero-copy memory provides quicker data exchanges between the CPU and GPU memories within the limits of the GPU memory availability. When dealing with large databases that exceed the capacity of the GPU memory, unified memory enabled via CUDA programming permits flexible, automated scheduling that surpasses the GPU memory threshold. This memory management strategy eliminates the need for manual memory manipulation, thereby improving overall efficiency.

Batch-based pipeline model. Another opportunity for optimization lies in the possibility of overlapping batch-to-batch transaction execution to achieve batch-to-batch pipeline execution. In LTPG, with inter-batch pipeline execution, we achieve concurrent execution of data transfer and transaction processing. While the nth batch of transactions is being executed, LTPG transfers the information of the $(n+1)$st batch of transactions to the GPU in preparation for execution, while the results of the $(n-1)$st batch of transactions are returned to CPU memory, and the aborted transactions of the $(n-1)$st batch are scheduled for re-execution in the $(n+2)$nd batch. This design takes advantage of the GPU's in-stream concurrency to achieve concurrent executions of computations and data transfers. The only drawback is that aborted transactions are not scheduled for direct execution in the next batch, but need to be scheduled for execution only two batches later. This may lead to an increase in latency for uncommitted transactions. However, in experiments, the overall latency of a batch of transactions is between $1ms$ and $100ms$ from the start of the transfer to the GPU to the end of the return of the transaction result to the CPUs. This latency increases with the increase of the database size and transaction batch. After the above optimization for high contention transactions, the transactions that cannot be committed in a single batch can be committed in fewer batches, and the latency is acceptable. Therefore, we achieve an overall performance improvement of 10%–20% at the cost of a few extra milliseconds of latency.

VI. Experiments

A. Experimental Settings

Benchmarks. We use two benchmarks TPC-C [1] and YCSB [7]. TPC-C aims to assess the performance of OLTP systems. It simulates complex workloads resembling an online retail environment with concurrent users performing various transactions. YCSB is to assess the performance of NoSQL and cloud-based databases. YCSB generates a workload that simulates typical operations in real-world applications, such as read, write, update, and delete operations.

Comparison systems. We compare with six CPU-based transaction processing systems, BOHM [10], PWV [9], Calvin [33], Aria [23], DBx1000 [37] and Bamboo [13], and two GPU-based transaction processing systems, GPUTx [14], and GaccO [4].

- **BOHM [10]** employs a two-step transaction execution process. In the first phase, primary key placeholders of the write set are inserted into a multi-version storage layer. In the second phase, each transaction reads a specific version for each key in its read set.
- **PWV [9]** decomposes each input transaction into fragments. A dependency graph is then computed to ensure data dependency and commit dependency requirements.
- **Calvin [33]** uses lock managers to grant read/write locks based on per-declared read/write-sets. Worker threads execute transactions when the lock manager acquires all locks for them.
- **Aria [23]** employs a two-step transaction execution process. In a read-write phase, it reads transactions from the current database snapshot and saves writes locally in a write set. In a commit phase, non-conflicting transactions are committed, while conflicting ones are rescheduled for the next batch based on their arrival time and TIDs.
- **DBx1000 [37]** is designed for multi-core environments and focuses on providing high-performance, scalable, and concurrent transaction processing. It aims to optimize database performance on modern hardware architectures with multiple CPU cores and supports both OLTP and OLAP workloads.
- **Bamboo [13]** is designed for multi-core environments and focuses on providing high-performance transaction processing on hotspots. Specifically, it modifies conventional two-phase locking while providing the same correctness guarantees.
- **GPUTx [14]** is a main memory DBMS that enables GPU-accelerated OLTP. When executing OLTP workloads, GPUTx computes a T-dependency graph and it manages

TABLE II
COMPARISON ON PART TPC-C (10^6TXS/S)

System	50-8[a]	50-16	50-32	50-64	100-8	100-16	100-32	100-64	0-8	0-16	0-32	0-64
DBx1000 [37]	2.64	2.62	2.24	1.73	1.81	1.66	1.39	1.15	3.87	7.14	10.04	8.35
Bamboo [13]	4.30	8.47	8.46	4.38	5.89	4.86	3.48	2.57	27.27	56.65	76.14	66.35
BOHM [10]	0.02	0.03	0.05	0.07	0.05	0.07	0.09	0.12	0.01	0.02	0.03	0.05
PWV [9]	1.27	1.51	1.64	1.53	0.93	1.13	1.23	1.16	3.09	3.51	3.66	3.37
Calvin [33]	0.39	0.40	0.39	0.31	0.32	0.29	0.27	0.21	0.88	1.01	1.06	0.91
Aria [23]	0.60	0.78	0.99	1.23	0.74	1.01	1.09	1.13	0.19	0.44	0.78	1.20
GPUTx [14]	0.02	0.01	0.10	0.09	0.17	0.10	0.07	0.05	0.68	0.54	0.42	0.26
GaccO [4]	16.06	15.80	15.55	15.29	13.72	13.12	12.75	12.65	134.92	134.47	135.21	135.04
LTPG	18.41	18.65	18.25	17.62	19.31	20.23	23.81	19.82	17.26	17.43	16.76	15.65

[a] We use the percentage of Neworder in a batch and the warehouse size as the experiment title.

a rank for each transaction in the batch. Transactions with the same rank can be executed simultaneously.

- **GaccO [4]** is a main memory DBMS for GPU-accelerated OLTP. For executing OLTP workloads, GaccO implements a novel scheme that splits the execution across the CPU and GPU. It constructs a dependency graph before transaction execution.

Both GaccO [4] and GPUTx [14] are not open-source. We obtained the source code of GaccO [4] from the authors and reproduced GPUTx [14]. For the other systems, we conduct experiments using their open-source codes.

Permanence metrics. We use Transactions Per Second (TPS), commit rate, and latency as performance metrics. TPS represents the number of completed transactions handled by the system in one second. Commit rate assesses the number of transactions successfully committed to the database. Latency measures the time taken for this batch of transactions to execute and return results. Both higher TPS and lower latency indicate higher transaction processing speed, while a higher commit rate refers to a higher success rate. For measuring TPS, we run 5,000 transaction batches back-to-back and report the total committed transactions and their execution times to eliminate randomness.

Additional settings. We implement pre-compiled, stored procedures using CUDA C++ to handle one-time and short transactions. A client interacts with the system by setting different parameters. Each table in the system has a primary key and attributes, using primary and secondary hash tables for indexing. Support for range queries is required for OrderStatus, StockLevel, and Delivery transactions in TPC-C. However, we can only predefine the primary key values of query items for the range queries in TPC-C, due to the nature of hash tables. LTPG can be *readily extended to support range queries*, by integrating indexing, such as B-trees and bitmap indexes. We aim to explore this in our future work. The experiments focus on combinations of Neworder and Payment transactions. This is because they make up almost 90% of the full TPC-C and because most of the systems we compare with only offer support for these two transaction types. 100% Neworder transactions have more operations and fewer conflicts. 100% Payment transactions have fewer operations and much more conflicts. The combination of 50% Neworder and 50% Payment transactions represents a moderate-contention load. All attributes in tables are set to integer type because CUDA does not support strings at present. We use a server with Ubuntu 22.04.1, CUDA-12.0, 8 Nvidia RTX A6000 GPUs, 2 Intel(R) Xeon(R) Gold 6326 CPUs @ 2.90 GHz and 768 GB RAM. We only use one A6000 GPU for the GPU-based systems. We schedule 30 CPU cores for the CPU-based systems. The memory we use is located in the same NUMA region. The source code of LTPG is available[1].

B. Comparison on TPC-C

For the mix of Neworder and Payment transactions processed exclusively on the GPU, as shown in Table II, LTPG exhibits a higher transaction throughput of approximately 18 million TPS. In comparison to other state-of-the-art systems, GaccO [4]–the most recent transaction processing system operating on GPUs–is capable of a throughput of approximately 15.5 million TPS. Moreover, it displays good performance for 100% Payment transactions with high conflicts. It is because the exchange operation optimization tailed for high-contention transactions in GaccO [4] greatly enhances parallel execution in heavy contention environments. GPUTx [14] achieves its best performance at about 10 thousand TPS. Aria [23], the latest deterministic database, achieves its highest throughput of about 0.8 million TPS on a standalone system. Meanwhile, DBx1000 [37], utilizing the TicToc [38] concurrency control mechanism, yields a throughput of approximately 5 million TPS. Bamboo [13] achieves 8 million TPS. BOHM [10], PWV [9], and Calvin [33] achieve 50 thousand, 1.5 million, and 40 thousand TPS. LTPG increases throughput by about $1.2\times$ over GaccO for mixed workloads and $1.5\times$–$1.9\times$ for 100% Neworder workloads. The higher TPS on mixed workloads and 100% Neworder workloads of LTPG are due to its use of deterministic optimistic concurrency control and its use of optimization strategies tailored for concurrent execution on GPUs. The deterministic optimistic concurrency control eliminates the need for LTPG to spend time constructing read and write sets and dependency graphs. The optimizations allow LTPG to complete the execution of a single batch transaction with lower time consumption. Overall,

[1]https://github.com/Kevin-W34/LTPG

TABLE III
PROCESSING CAPABILITY OF LTPG (10^6TXS/S)

Batch size	50-8[a]	50-16	50-32	50-64	100-8	100-16	100-32	100-64	0-8	0-16	0-32	0-64
2^8	1.62	1.22	0.82	0.47	3.64	1.99	1.11	0.59	1.34	1.02	0.76	0.47
2^{10}	5.33	4.10	2.71	1.74	11.40	7.07	4.05	2.23	3.90	3.27	2.32	1.58
2^{12}	13.00	11.04	8.24	5.16	17.98	17.66	11.41	7.57	10.12	8.43	6.07	3.78
2^{14}	18.21	17.83	15.90	10.51	14.92	17.26	23.81	17.78	16.51	15.02	12.29	9.50
2^{16}	14.56	16.66	18.15	17.42	5.71	9.39	13.44	16.12	16.64	17.35	16.76	14.20

[a] We use the percentage of Neworder in a batch and the warehouse size as the experiment title.

TABLE IV
EVALUATION OF AVERAGE PER-BATCH TRANSACTION LATENCY (μS)

System	8/8,192[a]	8/65,536	64/8,192	64/65,536
LTPG	357, 43[b]	1,794, 293	725, 46	2,621, 304
GaccO	1,270, 375	7,772, 2,627	1,292, 376	14,672, 3,076

[a] Warehouse size and batch size.
[b] per-batch transaction latency and data transmission latency.

TABLE V
EVALUATION OF OVERHEAD OF TRANSACTION READ/WRITE-SET.

batch size (Txns)	1,024	16,384	65,536
time cost (μs)	25–30	108–118	295–305

in real-world applications, where low-to-moderate conflicts are common, LTPG achieves better performance.

C. Processing Capability on GPUs

We study the batch processing capability on the GPU side using a balanced 50/50 workload combination. We use two main workloads from TPC-C: Neworder and Payment. Table III reports the results. All results include the CPU-to-GPU data transfer overhead. Moreover, we increase the database size from 8 to 64. We scale up the batch size from 2^8 to 2^{16} and use all optimizations in LTPG. We report transaction throughput in Table III and latency in Table IV.

We see that LTPG provides an aggregated throughput of over 17 million transactions per second in the 8-warehouse scenario. When the number of warehouses is less than 32, the throughput of LTPG increases as the number of warehouses grows. This is because the increase in normally committed transactions per batch outweighs the extra system overhead caused by the increase in the number of warehouses. The decrease in throughput from 32 to 64 warehouses shows that the additional overhead from GPU memory at this point no longer can be outweighed by the rise in transaction commits within a single batch, hence we see a decreasing trend in the aggregate transaction aggregation throughput. Overall, LTPG executes different transaction types with high throughput.

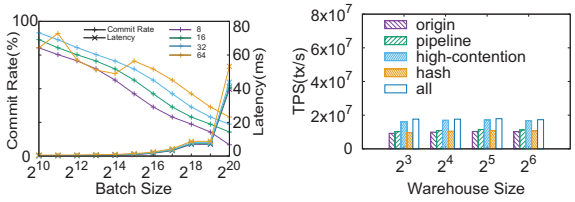
Second, our adaptation to GPU execution conditions, where we partition transactions into classes of transactions, allows different kinds of transactions to be executed simultaneously within the GPU without branching divergence. Next, LTPG does not suffer from severe aborts of Payment transactions in the 8-warehouse case. This is attributed to the effectiveness of LTPG's optimization scheme in handling high-contention transactions. Finally, LTPG has significant advantages in terms of latency in Table IV. Regardless of the database size or transaction batch size, LTPG has distinct advantages over GaccO. This allows LTPG to re-execute aborted transactions more quickly. In terms of data transfer costs, LTPG has significantly lower overhead than GaccO. This allows LTPG to have relatively more time for GPU computing, thereby improving GPU utilization.

We study the performance of all methods on 100% Neworder and 100% Payment transactions. Table III reports the results. LTPG achieves a maximum transaction throughput of 24 million on 100% Neworder transactions. Next, the maximum transaction throughput of LTPG is only 17 million on 100% Payment transactions. This indicates the strength of the deterministic optimistic concurrency control utilized by LTPG and the acceleration provided by GPUs, on low-contention workloads and on high load with more contention (transactions that contain many operations). However, with more contention, the transaction performance exhibited by LTPG is increasingly affected by inter-transaction conflicts, even with low load (transactions that contain few operations).

We also report the overhead of copying transaction read/write-set in Table V. When using the transmission method of transaction read/write-set, LTPG introduces additional latency ranging from 25μs to 300μs, where the latency is proportional to the transaction batch size. Compared to CPU-based systems, LTPG only introduces the latency of transferring data from the GPU to the CPU. As the CPU can asynchronously merge snapshots, it does not impact the transaction processing efficiency on the GPU.

D. Effectiveness of LTPG

Latency and commit rate. We report the latency and the commit rate with a 50/50 workload of Neworder and Payment transactions. We report the end-to-end latency for all transactions in a batch, i.e., the time from when parameters are transferred to the time when the batch results are available to the CPU. We report the percentage of committed transactions in a batch. In Fig. 6(a), the latency of a batch of transactions is between 300μs and 8ms when the transaction throughput is stable at 10 million transactions per second. Fig. 6(a) also shows the commit rate of a single batch of transactions is stable between 50% and 75% when the transaction throughput is stable at 10 million transactions per second. It can be seen that the LTPG system performs well in terms of both latency and commit rate. There is no deterioration in either metric, which is due to the large batch of transactions, the



(a) Commit rate and Latency of LTPG at different batch sizes.

(b) Transaction throughput at different optimization scenarios.

Fig. 6. Other indicators and optimization effect.

deterministic optimistic concurrency control, and the GPU computing capabilities.

High-contention transaction optimization. We assess the impact of optimizing for high-contention transactions on batch transaction commits, using a configuration of 32 warehouses and a batch size of 16,384. Table VI shows the results."commit transactions" denotes the total number of committed transactions and the numbers of committed transactions of types: Neworder and Payment, within a batch. "commit rate" denotes the percentage of successful transactions, again showing the split between Neworder and Payment transactions. The results indicate clearly that the optimization improves the success rate and quantity of high-contention transactions. When a batch is treated as a whole, the overall success rate of transactions rises by 20%–30% compared to scenarios without high-concurrency transaction optimization.

The positive impact of the high-contention optimizations becomes even more significant when examining specific scenarios. As shown in Table VI, introducing high-concurrency optimizations enhances the batch commit rate from a mere 0.41% to an impressive 52.9%. This leap in performance is attributed to the row-level timestamp splitting. It ensures that access to non-popular attributes in rows with popular data is unaffected by the popular attributes, enhancing the transaction commit rate. Furthermore, the proposed delayed update strategy for transactions permits popular attribute accesses in popular data rows to be committed successfully within a batch, given that there are no conflicts with other data items.

The impact of optimizations. We consider experiments using a 50/50 workload of Neworder and Payment transactions to evaluate the impact of the different optimizations of the GPU-based transactional batch processing. We report the overall performance achieved when introducing the different enhancements one by one into an initial, unenhanced version of LTPG.

The effect of inter-transaction pipeline execution on the cost of overlapping data transfers is considered first. Fig. 6(b) shows pipeline execution of different batches of transactions improves the overall transaction throughput between 10% and 15%. The benefit is lower than expected, which is mainly due to the time consumption structure of LTPG executions. The effect of memory transfer hiding is not obvious in data transfer time. The optimization of high-contention transactions impacts LTPG significantly, resulting in an impressive speedup of 1.75×. This is an indication that the traditional OCC scheme faces more severe transaction aborts when exposed to high-contention. LTPG's combination of deterministic OCC and transaction batch processing exacerbates the issues in such cases. Our optimization scheme for high-contention transaction scenarios exploits numerous opportunities for GPUs to address these challenges effectively.

TABLE VI
EVALUATION OF COMMIT RATE (%) AND COMMIT TRANSACTIONS (TXS/S)

Database scale/ batch size	Has optimization	Commit transactions	Commit rate
32[a]/16,384	yes	11,566, 7,232, 4,334	70.6, 88.3, 52.9
32/16,384	no	7,265, 7,233, 32	44.3, 88.3, 0.41
32/4,096	yes	3,294, 1,983, 1,311	80.4, 96.8, 64.0
32/4,096	no	2,017, 1,985, 32	49.2, 96.9, 1.56
8/16,384	yes	9,244, 5,192, 4,052	56.4, 63.4, 49.4
8/16,384	no	5,199, 5,191, 8	31.7, 63.4, 0.10
8/4,096	yes	2,892, 1,806, 1,085	70.6, 88.2, 53.0
8/4,096	no	1,817, 1,809, 8	44.4, 88.3, 3.91

[a] We use warehouse size to represent the scale of the database.

Hash table optimization. In the case of hash table optimization, we see a notable 5%–10% improvement in overall performance. This enhancement is primarily attributed to reducing the serial wait time between CUDA atomic operations, which contributes significantly to overall performance. However, these serial waits occur only during the transaction execution phase and not throughout the entire phase of GPU-accelerated transaction processing. We believe further optimizations in this area may result in additional improvements.

We study the impact of the hash bucket size on the efficiency of executing atomic operations — see Table VII. Each cell has two triplets corresponding to the case of $s_u = 1$ and $s_u = 32$. Each triplet represents the combined latency for marking and reading the TID, the latency for only marking the TID, and the latency for only reading the TID, respectively. For example, with a thread scale of 512×512 and a hash table size of 32, a bucket size of 1 results in a total latency of 83μs for marking and reading the TID. Out of this, marking takes up 76μs, and reading takes 7μs.

As observed, although the reading times between the two bucket sizes do not vary substantially, marking times (and consequently total times) are notably longer with a standard-sized bucket ($s_u = 1$) compared with a large-sized one ($s_u = 32$). This is because (i) data writes to global memory in GPUs are inherently time-consuming, and the presence of serialized atomic operations during writing further increases the time needed; and (ii) in contrast, large-sized buckets distribute serialized atomic operations across multiple slots within a bucket, which reduces serialization time substantially, leads to a noticeable improvement in performance. We also report the conditions of standard-sized buckets and large-sized buckets in memory, as shown in Table VIII. It is evident that with the growth of the warehouse size, the occupancy of large-sized buckets in the memory of the hash table used for marking conflicts remains extremely low. This confirms that our dynamic hash bucket optimization strategy provides

TABLE VII
COMPARISON OF LATENCY (μS) BETWEEN STANDARD-SIZED HASH BUCKET ($s_u = 1$) AND LARGE-SIZED HASH BUCKET ($s_u = 32$).

Grid×Block	hash table = 1	hash table = 32	hash table = 512
1,024×1,024	(652,638,14), (117,105,12)	(685,653,32), (128,112,16)	(675,660,15), (123,108,15)
512×512	(175,167,8), (38,31,7)	(83,76,7), (44,37,7)	(67,60,7), (37,30,7)

TABLE VIII
MEMORY OCCUPANCY RATIO OF STANDARD HASH TABLES AND LARGE HASH TABLES (%)

Bucket size	8[a]	16	32	64
large	0.053	0.054	0.054	0.055
standard	99.947	99.946	99.946	99.945

[a] Warehouse size.

TABLE IX
UNIFIED MEMORY TIME CONSUMPTION (μS)

database scale	execution phase	check conflicts phase	writeback phase
32[a]	589	37	69
512[a]	508	62	119
1,024[b]	494	53	113
2,048[b]	3,211	618	4,450

[a] Zero-copy memory mode.
[b] Unified memory mode.

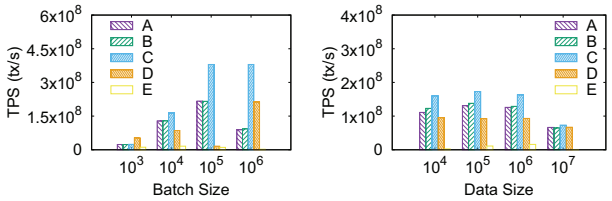
advantages in terms of batch transaction execution latency at a low memory cost.

Unified memory optimization. To accommodate oversized databases, LTPG employs selective memory mode adjustments to execute transactions efficiently. While unified memory performs automatic memory transfers implicitly, these processes contribute substantially to the total transaction processing time, thereby causing a notable reduction in LTPG's overall transaction throughput. Enhancing data transfer efficiency to address the requirements of oversized databases is a promising research direction. We evaluate the unified memory time consumption and report the findings in Table IX. We use a batch size of 16,384. With zero-copy memory, the time spent on the three stages is relatively small. However, when using unified memory, there is a significant increase in the time consumption of the three stages. This is primarily due to the frequent occurrence of page faults in the system. As mentioned before, the system triggers frequent data page swaps, increasing the time required for each stage considerably.

E. Performance on YCSB

We conduct experiments to evaluate the performance of LTPG with different settings on YCSB. We use a Zipfian data distribution with $\alpha = 2.5$ to represent a high-contention scenario and set each transaction to contain 10 operations. The data cardinality varies from 10^4 to 10^7.

Fig. 7 shows the performance of LTPG when using the full set of workloads of YCSB [7], including Update heavy (A), Read heavy (B), Read only (C), Read latest (D), and Scan in short ranges (E). We see that the workload involving scanning (E) performs the worst and that the read-only workload (C) performs the best among the five workloads. The reason is that



(a) Data Size = 10^6. (b) Batch Size = 10^4.

Fig. 7. Transaction throughput of different batch sizes and different data sizes with full YCSB.

range scans on GPUs necessitate specialized index and data structures, which impacts the overall performance adversely compared to read-only workloads.

F. Discussion

The experiments covered here show that the sweet spot for LTPG is scenarios with medium to high loads and less frequent access to popular data. In such cases, LTPG achieves high throughput. The adaptive warp division optimization and dynamic hash table optimization of LTPG are effective at improving the transaction processing performance. However, when there is a higher frequency of popular data accesses in a transaction batch, LTPG may experience more transaction aborts. In such situations, the high-contention transaction optimization scheme of LTPG is effective at reducing the transaction abort rate.

VII. CONCLUSION

We presented LTPG, a novel GPU-based transaction processing system tailored for vectorized execution. Unlike existing systems, LTPG leverages parallelization within warps to exploit the parallelism of GPUs. Conflict log utilizes dynamic hash buckets to reduce serialized executions of CUDA atomic operations. Moreover, we introduce a set of optimizations designed for high contention scenarios. We report on an experimental study showing that LTPG is able to outperform contemporary multi-core CPU OLTP systems by up to 7× and to achieve improvements up to 1.9× over leading GPU-based OLTP systems on the TPC-C benchmark in terms of throughput. In terms of single-batch latency, LTPG reduces latency by 44% to 72% compared with the state-of-the-art GPU-based system.

Acknowledgment. This work was supported by National Key R&D Program of China (2023YFB4503600), the National Natural Science Foundation of China (U23B2019, 62072083, 62372097), and the Fundamental Research Funds of the Central Universities (N2216017).

REFERENCES

[1] TPC Benchmark C., 2010.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[3] N. Boeschen and C. Binnig. Galop: Towards a GPU-accelerated OLTP DBMS. In *DAMON*, pages 9:1–9:3, 2021.

[4] N. Boeschen and C. Binnig. Gacco - A GPU-accelerated OLTP DBMS. In *SIGMOD*, pages 1003–1016, 2022.

[5] S. Breß. Why it is time for a hype: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proc. VLDB Endow.*, 6(12):1398–1403, 2013.

[6] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906, 2016.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[8] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[9] J. M. Faleiro, D. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, 2017.

[10] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, 2015.

[11] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, pages 15–26, 2014.

[12] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: query co-processing using graphics processors. In *SIGMOD*, pages 1061–1063, 2007.

[13] Z. Guo, K. Wu, C. Yan, and X. Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *SIGMOD*, pages 658–670, 2021.

[14] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, 2011.

[15] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *EuroSys*, pages 31:1–31:16, 2016.

[16] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.

[17] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[18] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, pages 1675–1687, 2016.

[19] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu. A GPU accelerated update efficient index for knn queries in road networks. In *ICDE*, pages 881–892. IEEE Computer Society, 2018.

[20] C. Li, Y. Gu, J. Qi, and G. Yu. Parallel skyline processing using space pruning on GPU. In *CIKM*, pages 1074–1083. ACM, 2022.

[21] T. Li, L. Chen, C. S. Jensen, and T. B. Pedersen. TRACE: real-time compression of streaming trajectories in road networks. *Proc. VLDB Endow.*, 14(7):1175–1187, 2021.

[22] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD*, pages 21–35, 2017.

[23] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A fast and practical deterministic OLTP database. *Proc. VLDB Endow.*, 13(11):2047–2060, 2020.

[24] S. Mu, S. Angel, and D. E. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *EuroSys*, pages 40:1–40:16, 2019.

[25] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS*, pages 263–273, 1999.

[26] N. Narula, C. Cutler, E. Kohler, and R. T. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[27] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[28] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2):145–156, 2012.

[29] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, 2014.

[30] K. Ren, A. Thomson, and D. J. Abadi. VLL: a lock manager redesign for main memory database systems. *VLDB J.*, 24(5):681–705, 2015.

[31] V. Rosenfeld, S. Breß, and V. Markl. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.*, 55(2):11:1–11:38, 2023.

[32] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. In *ICDE*, pages 698–709, 2019.

[33] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[34] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32. ACM, 2013.

[35] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, 2016.

[36] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD*, pages 1643–1658. ACM, 2016.

[37] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, 2014.

[38] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*, pages 1629–1642, 2016.

[39] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.