GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures

JIAPING CAO*, The Hong Kong Polytechnic University, China
LE XU*, The Hong Kong Polytechnic University, China
MAN LUNG YIU, The Hong Kong Polytechnic University, China
JIANBIN QIN, Shenzhen University, China
BO TANG†, Southern University of Science and Technology, China

Hash tables are widely used to support fast lookup operations for various applications on key-value stores and relational databases. In recent years, hash tables have been significantly improved by utilizing the high memory bandwidth and large parallelism degree offered by Graphics Processing Units (GPUs). However, there is still a lack of comprehensive analysis of the lookup performance on existing GPU-based hash tables. In this work, we develop a micro-benchmark and devise an effective and general performance analysis model, which enables uniform and accurate lookup performance evaluation of GPU-based hash tables. Moreover, we propose GPH, a novel GPU-based hash table, to improve lookup performance with the guidance of the benchmark results from the analysis model devised above. In particular, GPH employs the perfect hashing scheme that ensures exactly 1 bucket probe for every lookup operation. Besides, we optimize the bucket requests to global memory in GPH by devising vectorization and instruction-level parallelism techniques. We also introduce the insert kernel in GPH to support dynamic updates (e.g., processing insert operations) on GPU. Experimentally, GPH achieves over 8500 million operations per second (MOPS) for lookup operation processing in both synthetic and real-world workloads, which outperforms all evaluated GPU-based hash tables.

CCS Concepts: • **Theory of computation → Bloom filters and hashing**; **Data structures design and analysis**; **Parallel algorithms**; • **Computing methodologies → Graphics processors**.

Additional Key Words and Phrases: GPU, hash table, perfect hashing, lookup throughput, memory-bound optimization, vectorization, instruction-level parallelism, dynamic updates, micro-benchmarking

ACM Reference Format:
Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 165 (June 2025), 26 pages. https://doi.org/10.1145/3725406

*Both authors contributed equally to this research, this work was done when Jiaping Cao and Le Xu were research assistants at Southern University of Science and Technology.
†The corresponding author is Dr. Bo Tang.

Authors' Contact Information: Jiaping Cao, The Hong Kong Polytechnic University, Hong Kong, China, csjcao1@comp.polyu.edu.hk; Le Xu, The Hong Kong Polytechnic University, Hong Kong, China, le-simon.xu@connect.polyu.hk; Man Lung Yiu, The Hong Kong Polytechnic University, Hong Kong, China, csmlyiu@comp.polyu.edu.hk; Jianbin Qin, Shenzhen University, Shenzhen, Guangdong, China, qinjianbin@szu.edu.cn; Bo Tang, Southern University of Science and Technology, Shenzhen, Guangdong, China, tangb3@sustech.edu.cn.

ACM 2836-6573/2025/6-ART165
https://doi.org/10.1145/3725406

Table 1. GPU-based hash table category

Solution category	GPU-based hash table	Lookup probe guarantee	Lookup throughput
Separate chaining	SlabList [14]	no	low
Open addressing	WarpCore [40]	no	medium
Cuckoo hashing	DyCuckoo [46]	O(1)	medium
Perfect hashing	**Our GPH**	**1**	**high**

1 Introduction

A hash table is a fundamental data structure in many data-intensive applications. In particular, it is widely used in key-value stores, e.g., Redis [10], Memcached [31], and MegaKV [55]. Besides, hash tables are core components of hash join algorithms [12, 25, 35] in relational database management systems (RDBMS).

In general, the hash table supports lookup, insert, and delete operations on key-value pairs. The workloads of the hash tables in the data-intensive applications are typically read-intensive [22–24], i.e., these workloads have a large proportion of lookup requests. Hence, many prior studies [24, 27, 32, 45, 50, 54] accelerate the lookup performance of hash tables by utilizing various techniques, e.g., guaranteeing the number of probes for each lookup operation, in CPUs. However, the data access pattern of different lookup operations in a hash table is random as the lookup keys are quite different. As a result, the lookup throughput of the hash table is limited by the memory bandwidth of the CPUs [55].

Consequently, exploiting the high memory bandwidth and large parallelism degree of Graphics Processing Units (GPUs) to accelerate the performance of hash tables has been studied [11, 22, 39, 43, 46, 55] in the literature. However, with the well-known GPU characteristics, the lookup performance of these GPU-based hash tables still varies significantly as their hash collision resolution mechanisms are different. In particular, the hash collision resolution mechanism of these GPU-based hash tables can be classified into 4 categories. We summarize each category with its representative approach in Table 1.

(I) Separate Chaining. The GPU-based hash tables [14, 47] in this category employ linked lists to store the values with the same hash key. Thus, they can dynamically allocate space for the key-value pairs to resolve hash collisions. SlabList [14] is a representative approach in this category. Obviously, it does not offer $O(1)$ probe guarantee as the length of the linked list can be arbitrarily large, which could result in a low lookup throughput.

(II) Open Addressing. It uses alternative entries to store the values when the hash collision occurs [39, 40, 42]. WarpCore [40] is the state-of-art GPU-based hash table in this category. It also does not guarantee $O(1)$ probes for every lookup operation as it may access a series of alternative entries. In general, its lookup throughput is slightly better, but could still suffer from a large number of probes on a series of alternative entries.

(III) Cuckoo Hashing. It is a special type of GPU-based hash table in the open addressing category as it employs eviction chain insert algorithm to guarantee $O(1)$ probes for every lookup operation. CUDPP [11] and DyCuckoo [46] are widely used approaches in this category. Their lookup throughput is generally better as they guarantee $O(1)$ probes for each lookup operation.

(IV) Perfect Hashing. It guarantees that each lookup operation requires only 1 probe due to the carefully designed hash function, which is widely used on CPUs [24, 27, 32, 54]. However, to the best of our knowledge, there is no GPU-based implementation for workloads with dynamic updates (e.g., a small proportion of insert and delete operations on GPU).

Even though existing GPU-based hash tables could be qualitatively classified based on the lookup throughput as these mechanisms offer different lookup probe guarantees, it is challenging to compare the existing GPU-based hash tables quantitatively as the actual lookup performance is determined by: (i) the practical probing cost of each GPU-based hash table and (ii) the utilization of the characteristics of GPU architecture in each of them. Thus, it motivates us to propose an efficient and general lookup performance analysis model for GPU-based hash tables, which takes both (i) and (ii) into account w.r.t. the end-to-end lookup throughput. There is still a lack of a GPU-based hash table lookup performance model in the existing research that meets the above criteria.

In this work, we devise an effective-and-generic lookup performance analysis model for GPU-based hash tables. It effectively quantifies the characteristics of GPU architectures (e.g., parallelism degree, memory coalescing) and the lookup probing cost of various GPU-based hash tables by the measurable metrics on GPU. Our proposed model not only can accurately and uniformly analyze the practical lookup performance of GPU-based hash tables, but also can be used to guide the design of novel GPU-based hash tables.

Hence, we further analyze existing GPU-based hash tables with our proposed model. Inspired by the observations and conclusions from the analyzed results, we devise GPH, which consists of the lookup kernel and the insert kernel, and achieves the highest lookup throughput among all these GPU-based hash tables. In particular, our GPH follows the perfect hashing scheme, which guarantees exactly one probe for each lookup operation (see the last row in Table 1). Since it is not trivial to adapt the CPU perfect hashing tables to GPU as they lack optimized GPU resource utilization and dynamic updates, we propose a new GPU-based perfect hashing schema in GPH to address these challenges. It maximizes the active warps in GPU for parallel lookup operation execution. In addition, we introduce Bucket Requester, a module in GPH lookup kernel, which utilizes the vectorization and instruction-level parallelism techniques to efficiently request buckets in global memory. GPH also devises the insert kernel to insert key-value pairs and resolve bucket overflow in parallel on GPU to support dynamic updates, which cannot be done by existing GPU-based perfect hashing approaches.

In summary, the technical contributions of this work are:

- We propose an effective-and-generic lookup performance analysis model with insightful observations on the results of micro-benchmark experiments, which reveals the lookup performance bottleneck of existing GPU-based hash tables and sheds light on improving them by devising novel GPU-based hash tables.
- We propose GPH, a novel GPU-based hash table, to improve the lookup performance by adapting the perfect hashing scheme. We also devise the insert kernel to enable parallel insertions in GPH. To the best of our knowledge, this is the first time such functionality has been supported by GPU-based hash tables in the perfect hashing category.
- We conduct extensive experiments on both synthetic and real-world datasets to demonstrate the superiority of GPH. GPH outperforms existing GPU-based hash tables in various cases and achieves 1.70× to 2.78× faster in lookup throughput w.r.t. the state-of-the-art GPU-based hash tables.

The rest of this paper is organized as follows. Section 2 provides the preliminaries of hash tables and elaborates on the representatives of GPU-based hash tables. Section 3 conducts the micro-benchmark on three GPU-based hash tables and proposes the lookup performance analysis model. Section 4 provides the overview of our proposed GPU-based hash table GPH. Section 5 elaborates on the design of GPH lookup kernel and highlights its key techniques to improve the performance. Section 6 discusses the GPH insert kernel. Section 7 verifies the efficiency of our proposed method, GPH, by extensive experiments, and Section 8 concludes this work.

2 Preliminaries

In this section, we first introduce the definition of hash table in Section 2.1, and then elaborate on the lookup schemes of different GPU-based hash table variants in Section 2.2.

2.1 Hash Table

A hash table is a fundamental data structure to store key-value pairs. For every key-value pair (k, v), k is the key and v is the corresponding value or a reference to the value. Typically, the hash table is implemented as an array of entries. Each entry could be a slot or bucket. Specifically, a slot can store a (k, v) pair, and a bucket contains multiple consecutive slots. If the table has m entries, it employs a hash function $\mathcal{H} : K \rightarrow \{0, \ldots, m-1\}$ to map each key k from the key domain K to an entry. When two keys map to the same entry, i.e., $\mathcal{H}(k_1) = \mathcal{H}(k_2)$, the hash collision occurs as the underlying entry is unable to store both. The hash collision resolution mechanisms are different in different hash tables, which we will present shortly. In general, the hash table T supports the following fundamental operations:

- lookup(k): the lookup operation searches the given key k in the hash table and returns v if the pair (k, v) is in T, or $\emptyset$ otherwise.
- insert(k, v): the insert operation inserts a key-value pair (k, v) into the hash table T.
- delete(k): the delete operation removes the key-value pair (k, v) associated with the given key k from the hash table T.

Typically, hash tables are widely used to offer fast lookup operations [15, 22], e.g., in key-value stores [10, 30, 31, 55]. In the corresponding workload of these applications, the proportion of lookup operations is significantly higher than insert/delete operations. For example, the workload for the Facebook Social Graph [23] consists of 99.8% read requests and 0.2% update requests. These read requests are the lookup operations to find the objects and associations in the graph. In this work, we focus on the read-intensive workloads on hash tables, i.e., each workload includes a large proportion of lookup operations and a small proportion of insert/delete operations.

2.2 GPU-based Hash Table

For hash tables on CPUs, the lookup performance is limited. For example, it is almost impossible to support a key-value store system with a throughput of 1000 MOPS (i.e., million lookup operations per second) on modern CPUs as each lookup operation in CPU-based hash tables triggers several memory accesses, and each memory access takes 50 to 100 nanoseconds [55] in modern commodity CPUs, which is obviously higher than 1 nanosecond for the targeted 1000 MOPS throughput.

Fortunately, GPUs enjoy high memory bandwidth and parallelism degree. Many studies [11, 14, 39, 40, 46] have exploited these characteristics of GPUs to accelerate the lookup performance of hash tables in recent years. The key differences between GPU-based hash tables are underlying hash collision resolution mechanisms. Regarding the performance of lookup operations, the number of hash table probes for each lookup operation is determined by the utilized hash collision resolution mechanisms. The existing GPU-based hash tables could be classified into 4 categories by considering their underlying hash collision resolution mechanisms [44]. We employ the example in Figure 1 to illustrate the effect of different hash tables on the number of triggered probes by lookup(5).

Separate Chaining. Separate chaining stores (k, v) at entry $\mathcal{H}(k)$, as shown in Figure 1(a). To resolve the hash collision, each entry points to the head of a linked list. Each node of the linked list stores all the key-value pairs sharing the same $\mathcal{H}(k)$. SlabList [14] is a representative GPU-based hash table in this category. Figures 1(a) (i) and (ii) show the positive and negative lookup in this category, respectively. Moreover, separate chaining is well-suited for dynamic resizing of the hash

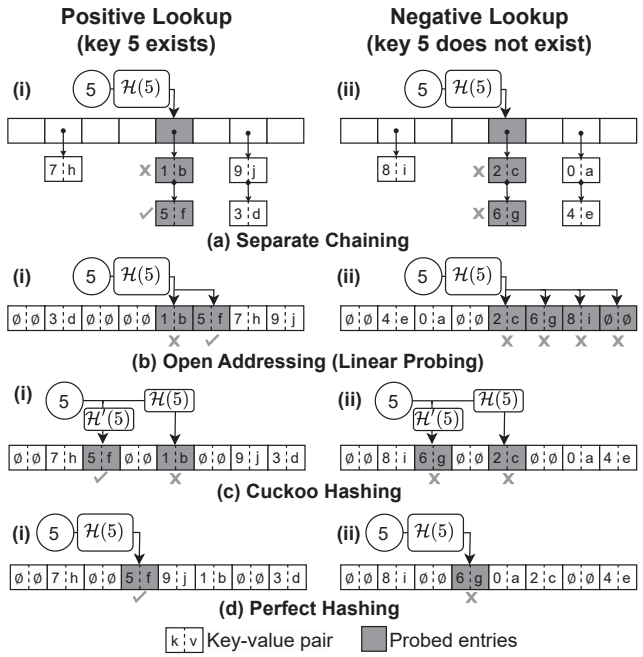


Fig. 1. Lookup probes of GPU-based hash tables

table as the size of the linked list can be extended. However, it does not provide an upper limit to the length of each list, thus the lookup probes cannot be guaranteed to be $O(1)$.

Open Addressing. Instead of allocating (k, v) in the linked list, open addressing uses alternative entries to resolve hash collisions. For lookup(k) operation, the entry $\mathcal{H}(k)$ is initially probed. If entry $\mathcal{H}(k)$ is full, a series of alternative entries that store (k, v) will be probed. For example, the linear probing method [20, 51] will probe the alternative entries $\mathcal{H}(k) + 1, \mathcal{H}(k) + 2, \ldots$ until either (k, v) is found (a.k.a., positive lookup) or a vacant entry is found (a.k.a., negative lookup), see the corresponding examples in Figures 1(b) (i) and (ii), respectively. Stadium Hashing [42] utilizes the double hashing method. When the entry $\mathcal{H}(k)$ is full, the alternative entries $\mathcal{H}(k) + 1 \cdot \mathcal{H}'(k), \mathcal{H}(k) + 2 \cdot \mathcal{H}'(k), \ldots$ will be probed. $\mathcal{H}'$ is an additional hash function. WarpCore [39, 40] employs a combination of linear probing and double hashing. Obviously, these hash tables do not guarantee $O(1)$ probes for every lookup operation since it may access other alternative entries.

Cuckoo Hashing. Cuckoo hashing is a special case of open addressing that guarantees the number of alternative entries is $O(1)$. In particular, it employs an eviction chain insert algorithm to resolve hash collisions and ensures that (k, v) can only reside within a configuration number of alternative entries. As shown in the examples in both Figures 1(c) (i) and (ii), lookup(k) only probes 2 entries, i.e., $\mathcal{H}(k)$ and $\mathcal{H}'(k)$. It is guaranteed that no other entries will be probed. Cuckoo hashing has become a widely used hash table variant. It has gained considerable attention in GPU-based hash tables [11, 22, 46, 55] as it offers a strong guarantee on the number of the lookup probes, i.e., $O(1)$. However, the number of lookup probes could be more than 1, e.g., the negative lookup operation in Figure 1(c)(ii) probed 2 entries.

Perfect Hashing. It ensures that only one entry $\mathcal{H}(k)$ can store (k, v) with a carefully designed hash function $\mathcal{H}$. Hence, every lookup operation (both positive and negative) probes exactly 1

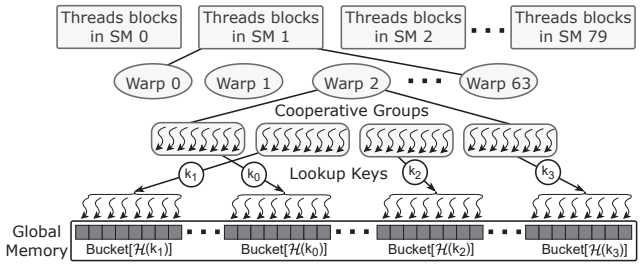


Fig. 2. Lookup operation processing procedure on GPU

entry, as illustrated in Figures 1(d) (i) and (ii). Generally, perfect hashing boosts lookup speed but costs more to update. It has been widely studied on CPUs [18, 21, 24, 27, 29, 32]. For example, DPH [29] is a variant of the CPU-based hash table with perfect hashing scheme, which supports dynamic data updates via a two-level hash scheme to guarantee exactly one lookup probe. Recently, MapEmbed and EEPH [24, 54] attempt to support dynamic perfect hashing by maintaining small auxiliary structures in fast memory to update the perfect hash function for accessing data in slow memory. However, none of these techniques on CPUs can be directly adapted to GPUs and take full utilization of the high memory bandwidth and parallelism degree. PHOBIC [37] contributes a GPU implementation to accelerate the minimal perfect hash function construction and uses it only on the CPU side. PSH [43] is an implementation of perfect hashing scheme on GPU, and its hash function $\mathcal{H}$ is pre-computed for the given static dataset to guarantee there is no hash collision. However, it only supports lookup operations on GPU and does not support insert and delete operations on GPU. It needs to reconstruct the PSH hash table on CPU to handle updates.

2.3 Other Relevant Studies

There are also many other techniques [35, 42, 55, 56] that have been proposed in the literature for various applications, using hash tables as building blocks (e.g., building key-value stores and hash join algorithms). To support string keys on GPU-based hash tables, MegaKV [55] offloads the variable-length string keys to CPUs and uses a signature algorithm, the cyclic redundancy check (CRC32), to obtain short fixed-size signatures of string keys. The key signatures will be loaded to GPU memory over PCIe. The polynomial rolling hash function, introduced in [34], is well-suited for GPUs as each character in the string can be processed independently. GPH focuses on hash table operations with fixed-size key-value pairs (e.g., 8 bytes) on the GPU side and leaves the support for other data types (e.g., strings, objects) as future work.

3 Performance Modelling

In this section, we first introduce the hash table lookup processing procedure on GPU in Section 3.1, then conduct a micro-benchmark to evaluate the performance of different GPU-based hash tables in Section 3.2, next conclude the insightful observations from the measured metrics of the micro-benchmark experiments in Section 3.3, and last propose an effective-and-generic lookup performance analysis model of GPU-based hash tables in Section 3.4.

3.1 Lookup Processing on GPU

As elaborated in Section 2, the GPU-based hash tables utilize different hash collision resolution mechanisms. However, the lookup operation processing procedure of them is similar as the key idea is exploiting the high memory bandwidth and large parallelism degree of GPU. Figure 2 depicts the

hash table lookup operation processing procedure on GPU. The hash table entries (i.e., buckets) are stored in the global memory of the GPU, and the bucket capacity (i.e., number of slots in a bucket) of different GPU-based hash tables are different. The thread blocks on all streaming multi-processors (SMs) process a subset of lookup operations in a batch. For example, all 80 SMs on NVIDIA V100S GPU will be used to process a batch of lookup operations. Each SM of a GPU contains on-chip shared memory and a fixed number of warps (e.g., it is 64 in NVIDIA V100S). All these warps in the same SM process lookup operations in parallel. Warp is the basic execution unit of GPU. Each warp has 32 threads and threads in the same warp execute the same instruction simultaneously, i.e., single-instruction multiple-threads (SIMT) microarchitecture of GPUs. To further improve the utilization of GPU by exploiting its memory coalescing feature, existing GPU-based hash tables [40, 46, 55] group several neighboring threads in the same warp into a cooperative group (see the middle of Figure 2) as the adjacent threads in the same cooperative group access contiguous locations in global memory, which is more efficient than performing random access. The threads in a cooperative group process the same lookup operation and access the different slots of the target hash table buckets in global memory. All the above GPU-based hash tables allocate different numbers of threads in their cooperative group to achieve good performance.

3.2 Micro-benchmark of GPU-based Hash Table

Until now, we distinguish the GPU-based hash tables by considering their underlying hash collision resolution mechanisms and highlighting their different settings (e.g., cooperative group size) during lookup operations. However, it is still unable to analyze the end-to-end lookup performance of different GPU-based hash tables. In other words, it does not have a fair performance comparison among these alternatives. In order to analyze the advantages and disadvantages of each GPU-based hash table, we conduct a micro-benchmark on them.

Micro-benchmark Setting. All experiments are conducted on the same machine with Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and an NVIDIA Tesla V100S GPU. The CUDA version is 12.5. Each table is built with 1.5 GB of global memory, and we insert 100 million randomly generated 8-byte packed key-value pairs. We perform 200 million lookup operations on each table, and the ratio of both positive and negative lookups is 50%. We use NVIDIA Nsight Compute CLI [9] to collect kernel execution metrics on GPU. The evaluated GPU-based hash tables use the default settings in their papers or codes.

Evaluated GPU-based Hash Tables. We evaluate three representative GPU-based hash tables: WarpCore [40], DyCuckoo [46] and CUDPP [11]. In particular, WarpCore resolves hash collision by the open addressing method. It is the state-of-the-art GPU-based hash table in this category [36]. Its cooperative group has 8 threads and accesses a bucket of 8 slots. Both DyCuckoo and CUDPP utilize cuckoo hashing method to address the hash collision. The cooperative group of DyCuckoo has 16 threads and accesses a bucket of 16 slots. CUDPP is the most widely used GPU-based hash table. Each thread independently accesses hash table entries with a single slot. In other words, each cooperative group of CUDPP only has 1 thread. Since separate chaining [14, 47] is designed to support efficient dynamic update operations rather than fast lookup operations (e.g., it works well for write-intensive workloads), its lookup performance is worse than WarpCore and DyCuckoo, as evaluated in [40] and [46]. We omit it in this work. We omit PSH [43] from the perfect hashing category as it only works on static data. We include our proposal GPH, a GPU-based dynamic perfect hashing table, in the micro-benchmark. We will introduce the technical details of our GPH shortly. The cooperative group of GPH has 4 threads and accesses a bucket of 16 slots.

Measured Execution Metrics. During the experiments, we collect the following execution metrics for analysis.

Table 2. Measured metrics of the GPU-based hash tables

	Category	AO (%)	EI (10^9 inst.)	CPI (cycles/inst.)	Time (ms)
WarpCore	Open Addressing	70.89	9.36	22.58	45.31
DyCuckoo	Cuckoo Hashing	86.87	18.69	24.78	88.10
CUDPP	Cuckoo Hashing	92.37	0.73	445.38	55.18
Our GPH	Perfect Hashing	**98.24**	**6.43**	**23.16**	**22.93**

- **A**chieved **O**ccupancy (*AO*) [3]: it measures the percentage of active warps in each SM.
- **E**xecuted **I**nstructions (*EI*) [8]: it is the total number of executed warp instructions.
- **C**ycles **P**er **I**nstruction (*CPI*) [7]: it shows the average cycles to execute a warp instruction.

3.3 Insightful Observations

The measured metrics and execution time cost are shown in Table 2. In this section, we summarize the insightful observations from the table.

Observation I: *AO* of WarpCore is obviously lower than that of DyCuckoo and CUDPP. As shown in Table 2, *AO* of WarpCore, DyCuckoo and CUDPP are 70.89%, 86.87%, and 92.37%, respectively. Moreover, the measured *AO* metric confirms that not all warps are active in these existing GPU-based hash tables when it is processing lookup operations.

Analysis of Observation I. The core reason for low *AO* is that some of the warps will be inactive after they complete their workload and wait for others. It is known as the tail effect [3]. In the evaluated existing GPU-based hash tables, each lookup operation can probe a different number of hash table entries. For example, the open addressing method WarpCore does not guarantee $O(1)$ probes. Thus, some WarpCore warps will probe many more entries than others. As a result, *AO* of WarpCore is the lowest (i.e., 70.89%) among the evaluated GPU-based hash tables. The cuckoo hashing methods DyCuckoo and CUDPP provide $O(1)$ guarantee on the number of lookup probes. Hence, *AO* of them are relatively high. Nevertheless, both DyCuckoo and CUDPP do not guarantee a consistent number of probes, e.g., DyCuckoo needs one or two lookup probes for each lookup operation.
Takeaway: *AO can be improved by guaranteeing the consistent number of lookup probes in the GPU-based hash table.*

Observation II: *EI* of existing GPU-based hash tables are quite different, but the processed lookup operations are the same. Specifically, the tested micro-benchmark provides the same workload for these three existing GPU-based hash tables, i.e., 200 million lookup operations. However, *EI* of existing GPU-based hash tables ranges from 0.73 billion to 18.69 billion instructions.

Analysis of Observation II. For a given lookup workload, the total number of lookup probes is fixed for each GPU-based hash table. As explained in Section 3.1, every lookup operation in the batch is processed by a cooperative group. With the single instruction multiple threads (SIMT) characteristic of GPU, *EI* is determined by the number of cooperative groups in a warp, i.e., the more groups in a warp, the less *EI*. For example, DyCuckoo has 2 cooperative groups in a warp, and it executes 18.69 billion instructions (i.e., *EI*) for the workload in the micro-benchmark. However, CUDPP only executes 0.73 billion instructions for the same workload as it has 32 cooperative groups in a warp, e.g., each thread in a warp is working independently.

Takeaway: *EI of the GPU-based hash table for a given workload is determined by the number of cooperative groups in a warp. The more groups in a warp, the less EI.*

Observation III: *CPI* of CUDPP is significantly larger than that of WarpCore and DyCuckoo. In the GPU-based hash table, *CPI* is a metric that reflects the memory access latency [16]. As shown in Table 2, *CPI* of CUDPP is high, i.e., 445.38 cycles per instruction. However, *CPI* of WarpCore and DyCuckoo are only 22.58 and 24.78 cycles per instruction, respectively.

Analysis of Observation III. The memory access latency of different GPU-based hash tables relies on the utilization degree of memory coalescing characteristic of GPU. The latency of CUDPP is the largest (i.e., the highest measured *CPI*) as the hash table entries in it are accessed in a random manner. However, WarpCore and DyCuckoo exploit memory coalescing techniques by using threads in cooperative groups to access the entries in consecutive memory. Thus, *CPI* of WarpCore and DyCuckoo is significantly smaller than that of CUDPP.
Takeaway: *Exploiting memory coalescing techniques reduces CPI of GPU-based hash tables.*

3.4 Lookup Performance Analysis Model

Even though we have three insightful observations about the micro-benchmark experiments on existing GPU-based hash tables, they cannot directly guide us to design a better GPU-based hash table. The core reason is that there is still a lack of a performance model between these measured metrics and end-to-end time cost. For example, *AO* of WarpCore is the lowest, but its end-to-end time cost is the best. Moreover, *EI* of CUDPP is the smallest, but *CPI* of CUDPP is the largest. Inspired by previous GPU performance analysis work [38], we propose the lookup performance analysis model of GPU-based hash table, see Equation (1), with these measured metrics, i.e., *AO*, *EI*, and *CPI*. It is an effective-and-generic model to (i) analyze the lookup performance bottleneck of GPU-based hash tables, and (ii) shed light on how researchers can improve GPU-based hash table performance by devising advanced techniques.

$$Time \propto \frac{EI \cdot CPI}{AO} \tag{1}$$

We next elaborate on the correctness of our proposed performance analysis model in Equation (1). The time cost of lookup operations is negatively correlated with *AO* as *AO* is a warp parallelism degree indicator, which means the number of active warps during the lookup batch processing. Moreover, $EI \cdot CPI$ is the total cycles we need to process the workload, which is correlated to the end-to-end time cost. Thus, calculating the exact time cost requires considering both $EI \cdot CPI$ (i.e., the total cycles) and *AO* (i.e., the warp parallelism degree) together. To further verify this conclusion, we compute the Pearson correlation coefficient [49] between $Time \cdot AO$ and $EI \cdot CPI$ via the measured metrics in Table 2. The result is 0.9993, which shows there is a strong linear relationship between these two values. As depicted in Figure 3, the plotted existing GPU-based hash tables lie around the linear line, which confirms the correctness of our proposed lookup performance analysis model in Equation (1).

In this work, we propose a novel GPU-based hash table GPH, which (i) improves *AO* by guaranteeing a consistent lookup probe for each lookup operation and (ii) reduces $EI \cdot CPI$ by proposing vectorization and instruction-level parallelism techniques. We will introduce the detailed technical contributions of GPH shortly. As shown in the last row of Table 2, *AO* of our GPH is the highest (98.24%) among all alternative approaches. Its $EI \cdot CPI$ is 148.91 billion, which is smaller than the 211.24 billion, 463.13 billion, and 325.91 billion values of WarpCore, DyCuckoo, and CUDPP, respectively. Thus, our proposal GPH has the smallest time cost (22.93ms) among all GPU-based hash tables in the micro-benchmark.

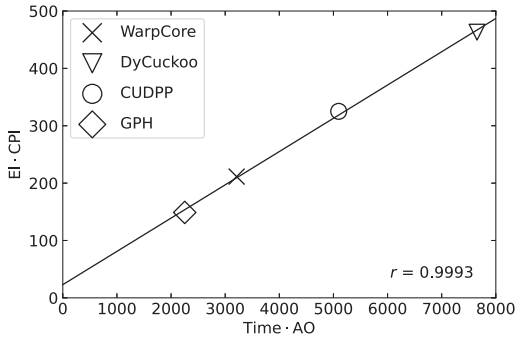


Fig. 3. The measured metrics and time cost regression

4 Our Proposal GPH

In this section, we introduce the architecture overview of GPH in Section 4.1, then present the data structure of GPH in Section 4.2.

4.1 GPH Architecture Overview

The first design choice of GPH is to choose a proper hash collision resolution mechanism. With Takeaway of Observation I and the lookup performance analysis model in Section 3, there is no doubt that the perfect hashing will be the answer. The key reason is that it guarantees exactly 1 probe for each lookup operation, thus resulting in high *AO*. However, it is not trivial to design a GPU-based hash table with perfect hashing scheme as (i) the state-of-the-art GPU-based hash table in perfect hashing category (e.g., PSH [43]) does not support dynamic updates; and (ii) adapting the existing CPU-based perfect hash tables [29, 32] to GPU is also challenging as they were not designed to fully utilize the characteristics (e.g., massive parallelism, memory hierarchy) of GPU. In this work, we propose GPH. To the best of our knowledge, it is the first GPU-based perfect hash table that supports both lookup and insert operations.

Figure 4 depicts the architecture overview of GPH, which consists of two major kernels: the lookup kernel and the insert kernel. Specifically, the lookup kernel of GPH processes batched lookup operations via **Lookup Groups**. Conceptually, it is equivalent to the cooperative groups in Figure 2. The memory access in each lookup operation consists of (i) reading the Cell Index in the shared memory of SM, and (ii) probing a hash table bucket in global memory. To improve the performance of the lookup kernel, we introduce a Bucket Requester module using vectorization and instruction-level parallelism techniques. Supporting dynamic updates in GPH is one of the major technical contributions of this work, as none of the existing GPU-based perfect hash tables support it. Specifically, we devise the insert kernel in GPH, which organizes threads into **Insert Groups** as basic units for insertion of key-value (k, v) pairs. The insert kernel of GPH divides the insert operations into two phases: (i) the Saturation Fill Phase, which uses a lock-free method to insert key-value pairs into GPH if it does not incur bucket overflow; (ii) the Refinement Phase, which introduces Extended Move to handle the bucket overflow caused by key-value pair insertions. In addition, we create a Workspace in shared memory to cache hash table buckets, reducing expensive global memory probes.

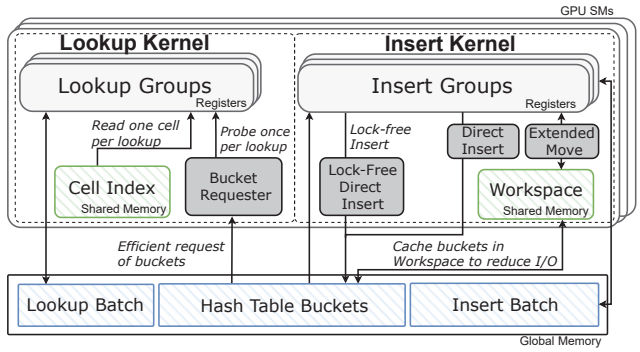


Fig. 4. The architecture overview of GPH

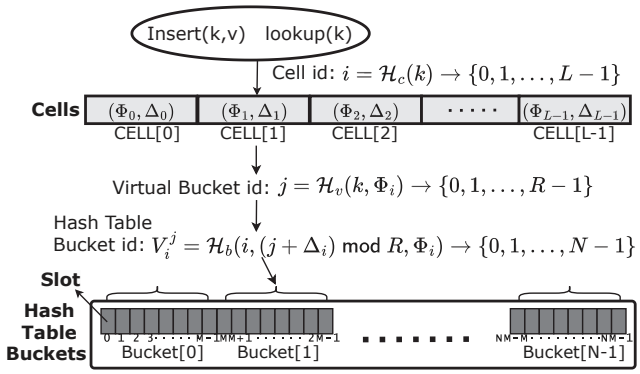


Fig. 5. Data structure in GPH

4.2 GPH Data Structure

As depicted in Figure 5, the data structure of GPH consists of (i) Cell Index and (ii) Hash Table Buckets. Cell Index is a cell array of size L, which is stored in GPU shared memory. Each cell in it stores a (Mapping Φ, Offset Δ) pair, which is the core of perfect hashing scheme. Hash Table Buckets are stored in GPU global memory, and GPH guarantees each lookup only incurs exactly 1 probe, i.e., accesses the global memory once. In particular, GPH has N Hash Table Buckets, each of which has M key-value slots. For each key or key-value pair to be looked up or inserted in GPH, it first maps to the cell i with (Φ_i, Δ_i) in Cell Index via the hash function $\mathcal{H}_c(k)$. Next, the hash function $\mathcal{H}_v$ takes the searched key k and Mapping Φ_i as input to compute the Virtual Bucket id j. The value range of j and Δ_i is in $[0, R-1]$, which means there are R Virtual Buckets in a cell. Last, the Hash Table Bucket id V_i^j of key k is computed via the hash function $\mathcal{H}_b$ with Cell id i, Virtual Bucket id j, Mapping Φ_i, and Offset Δ_i.

GPH is the first perfect hash table on GPU that supports both insert and lookup operations. The novel techniques and the advantages of GPH can be summarized in three key aspects: (i) it utilizes the fast shared memory for Cell Index and slow global memory for Hash Table Buckets on GPU to improve the lookup performance of GPH; (ii) it first proposes the (Mapping Φ, Offset Δ) pair in Cell Index to guarantee exactly 1 probe for each lookup and support dynamic updates in GPH; (iii) it introduces Virtual Buckets to handle the bucket overflow during key-value pair insertion. In particular, each cell maps to R Virtual Buckets, and it only needs to relocate the key-value

pairs within the R Virtual Buckets whenever bucket overflow occurs. In the following sections, we explain how to utilize these components to implement the lookup and insert kernels of GPH.

5 Lookup Kernel in GPH

In this section, we introduce the GPU lookup kernel of GPH. Section 5.1 describes the processing procedure of lookup kernel in GPH. Section 5.2 elaborates on the design of lookup groups and the Bucket Requester.

5.1 The Processing Procedure of Lookup Kernel

In the GPH lookup kernel, GPH organizes several consecutive threads in a warp to form a Lookup Group g to process the same lookup operation. The Lookup Group concept in GPH is similar to the cooperative group in existing GPU-based hash tables, and we will introduce the difference shortly. We illustrate the processing procedure of lookup kernel in GPH by the running example lookup(8) in Figure 6.

- **First, it computes Cell id i and accesses $CELL[i]$.** The Cell id $i = \mathcal{H}_c(8) = 4$, then it reads $\Delta_4 = 2$ and $\Phi_4 = 0$ in $CELL[4]$.
- **Second, it computes Virtual Bucket id j.** The Virtual Bucket id of the searched key 8 is computed by $j = \mathcal{H}_v(8, 0) = 2$.
- **Third, it computes the Hash Table Bucket id V_i^j.** With Cell id 4, Virtual Bucket id 2, and (Mapping Φ_4, Offset Δ_4), the corresponding Hash Table Bucket id is computed via $V_4^2 = \mathcal{H}_b(4, (2+2) \bmod 4, 0) = 3$.
- **Last, it accesses and checks $B[V_i^j]$ for the searched key.** $B[3]$ is accessed by the Bucket Requester module in GPH and returns (8,3) in $B[3]$ as the result of lookup(8) .

As stated in Section 4.2, GPH guarantees that lookup(k) only probes one Hash Table Bucket, i.e., $B[V_i^j]$, to determine the existence of (k, v) in the hash table. In other words, the key-value pair (k, v) can only be in $B[V_i^j]$, otherwise it does not exist in the hash table. This is the reason that GPH is categorized as perfect hashing and achieves high *AO* (see Takeaway of Observation I). However, accessing only one Hash Table Bucket for each lookup operation is still the most costly step in lookup kernel, because it includes accessing M slots located in the global memory of GPU. To further accelerate this step, we introduce the Bucket Requester module in Section 5.2, which is critical to improve the lookup performance.

5.2 Efficient Hash Table Bucket Access

The Hash Table Bucket access is the most expensive step in the lookup processing procedure in GPH. As discussed in Section 3, existing GPU-based hash tables typically leverage cooperative groups to address this challenge. The threads in a cooperative group process the same lookup operation and access the different slots of the target hash table bucket in global memory. This strategy facilitates efficient coalesced memory access, hence it reduces *CPI* in Equation (1), as Takeaway of Observation III shows.

An example of bucket request in the existing GPU-based hash table DyCuckoo is illustrated in Figure 7(a), where lookup kernel of the DyCuckoo uses the cooperative group of 16 threads to access a bucket of 16 slots via a warp instruction. However, there is a trade-off between the number of cooperative groups and the number of threads in each cooperative group as the total number of working threads in GPU is fixed. Thus, it may result in high *EI*, see Takeaway of Observation II. According to our performance model, a higher *EI* reduces the overall performance.

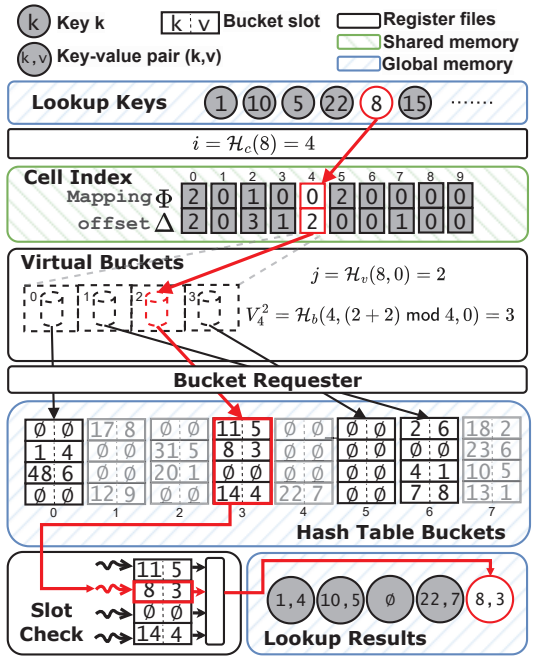


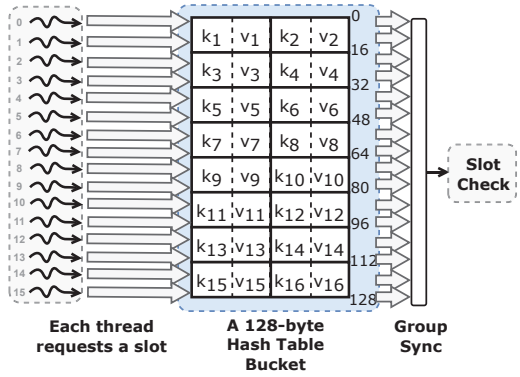
Fig. 6. GPH lookup kernel

To overcome this and minimize $EI \cdot CPI$ (see our performance analysis model), we devise the Bucket Requester module in GPH. The key idea of Bucket Requester is to efficiently request more buckets via each warp instruction. In particular, the design of Bucket Requester utilizes two techniques: vectorization and instruction-level parallelism (ILP).

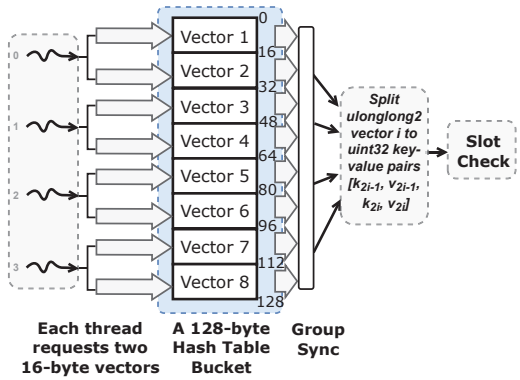
Algorithm 1 Setup configuration for bucket requester

```
1: def setup_bucket_requester_config():
2:     # max bytes requested per thread is 16 bytes
3:     max_trans_bytes = 16
4:     bytes_per_slot = sizeof(KV_TYPE)
5:     config.slots_per_thread = M div |g|
6:     # try to let each thread request full 16 bytes by regarding slots as a vector
7:     config.slots_per_vector=min(max_trans_bytes div bytes_per_slot,slots_per_thread)
8:     config.vectors_per_thread = config.slots_per_thread div config.slots_per_vector
9:     return config
```

Vectorization. In the bucket request methods of existing GPU-based hash tables, each thread typically accesses one slot, e.g., each thread accesses a slot in DyCuckoo, see in Figure 7 (a). However, if we use fewer threads in a group to request a bucket with the same size, each thread is assigned to access multiple slots, and this can be accelerated by exploiting the GPU vectorization technique [2]. In GPH, we utilize CUDA's official built-in vector types [5], which can interpret multiple contiguous slots of up to 16 bytes as a vector data type. For example, two adjacent 8-byte slots in a bucket can be interpreted as a 16-byte ulonglong2 vector. Thus, we can interpret several adjacent slots in a bucket as a vector, and threads can access a vector instead of a slot in a warp instruction.



(a) An example of existing approach of requesting a bucket



(b) An example of GPH Bucket Requester

Fig. 7. Before & after using Bucket Requester module to request a bucket from 16 threads. GPH only needs a 4-thread group to request a 128-byte bucket as efficiently as 16-thread group in traditional method

Algorithm 2 Bucket requester

```
1: def bucket_requester(B, bkt_id, config, lane):
2:     # each thread requests a vector per instruction
3:     V_TYPE = type of vector with config.slots_per_vector slots
4:     VB = reinterpret B to be array of V_TYPE vectors
5:     vectors = array of length config.vectors_per_thread in V_TYPE
6:     # unroll loop
7:     for i in range(config.vectors_per_thread):
8:         vectors[i] = VB[bkt_id * M + config.vectors_per_thread * lane + i]
9:     return vectors
```

To ensure that each thread accesses the maximum number of slots (i.e., 16 bytes in CUDA) in a warp instruction, Algorithm 1 determines the configuration, i.e., the slots for each thread to access, the slots in each vector, and the number of vectors that each thread should request. The sketch of Bucket Requester is depicted in Algorithm 2. Lines 3 and 4 reinterpret the memory region of Hash Table Buckets in the type of vectors. Line 5 allocates registers to hold the requested vectors. Lines 6

Algorithm 3 Slot check

```
1: def slot_check(vectors, k, config, group_mask):
2:     res_lane = -1
3:     # split the requested vector into key-value pairs
4:     slots = array of length config.slots_per_vector in KV_TYPE
5:     for i in range(config.vectors_per_thread):
6:         slots = split_vectors_into_kvs(vectors[i])
7:         for j in range(config.slots_per_vector):
8:             # ballot_sync gets bitmask of threads with key == k
10:            res_mask = ballot_sync(group_mask, slots[i].key==k)
11:            # ffs returns position of first set bit
12:            res_lane = ffs(resMask & -resMask) - 1
13:            if res_lane >= 0: break
14:        if res_lane >= 0: break
15:    return res_lane
```

to 9 request and return vectors consisting of multiple slots. The returned vectors are verified, which are split into slots (Line 6, Algorithm 3), and the slot check is performed for the lookup answer.

Instruction-Level Parallelism (ILP). In order to keep more threads active to execute instructions, the ILP feature of GPU allows threads not to stall after issuing memory requests but to keep executing other instructions. The threads will only stall at the instructions depending on the value not yet fetched from memory [53]. Thus, a thread in a GPU-based hash table does not stall immediately after requesting the Hash Table Bucket, but only stalls at the slot check which compares the slot value with the key. Thus, we utilize the ILP feature by issuing the memory requests to the Hash Table Bucket as many as possible before the slot check. Our strategy is to buffer all requested vectors and perform the slot check (Algorithm 3) at the end of the lookup operation. Although this strategy uses more registers for buffering the requested vectors, it allows all memory requests to be issued before any stall. Besides, we have observed that registers are sufficient for buffering vectors in most practical cases. Thus, all memory requests of vectors will first be issued in Algorithm 2, and then threads only stall at slot check in Algorithm 3.

Figure 7 (b) provides an example of Bucket Requester with vectorization and ILP techniques in the lookup kernel of GPH. It shows how a lookup group of $|g| = 4$ utilizes Bucket Requester to request a Hash Table Bucket with 16 slots, each 8 bytes in size. The buckets are interpreted as 8 vectors, each 16 bytes in size (ulonglong2), so that each vector contains two slots. Each thread issues requests to the two vectors before the thread stalls at any slot check. Finally, the thread splits the requested 2 vectors into 4 slots and performs the slot check. Compared to the existing approach that uses 16 threads per group in Figure 7(a), Bucket Requester uses only 4 threads per group. Thus, GPH can have more groups in a warp.

In GPH, we employ Bucket Requester for efficient bucket access, which enables GPH to use fewer threads in each lookup group to efficiently access all the slots in the bucket. We will verify GPH can access the buckets of the same size with far less *EI* and similar *CPI* compared to the existing GPU-based hash tables in Section 7. Interestingly, the techniques we have used in Bucket Requester are generic, and can be adapted to the existing GPU-based hash tables. We leave it as future work as it is out of the scope of this work.

6 Insert Kernel in GPH

Supporting dynamic updates on the GPU-based perfect hash table is challenging as the one probe property of perfect hashing is not easy to guarantee. In this section, we introduce the GPH insert

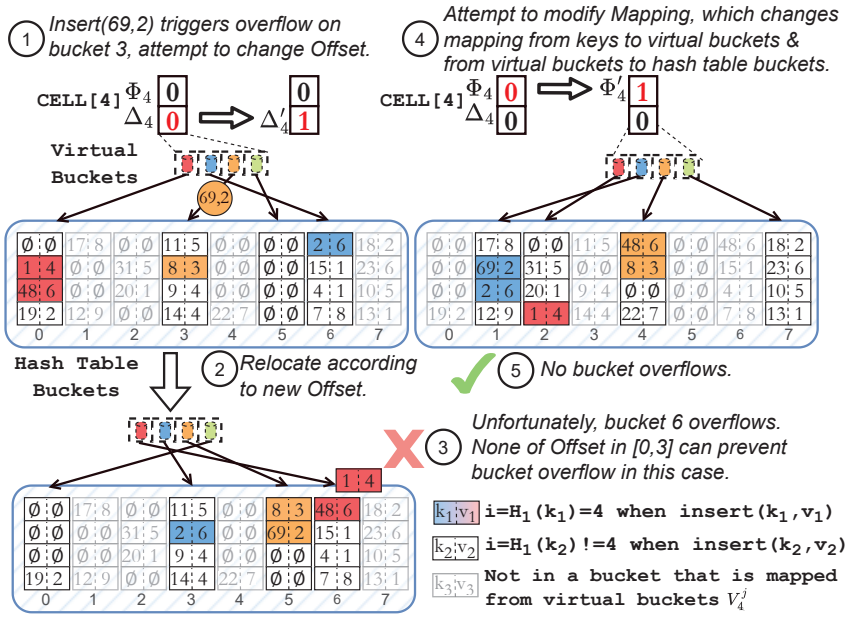


Fig. 8. An example of GPH Extended Move to resolve bucket overflow

kernel. In particular, Section 6.1 describes the insert processing procedure. Section 6.2 analyzes the theoretical guarantee for insertion in GPH. Section 6.3 presents how to support parallel insertions in GPH.

6.1 The Processing Procedure of Insert Kernel

The insert kernel organizes threads into the Insert Groups. To process insert(k, v), Insert Group h reads (k, v) and accesses a Hash Table Bucket $B[V_i^j]$, which is processed as the lookup kernel with searched key k. If there is a vacant slot in $B[V_i^j]$, h writes (k, v) to the slot in it, which is a Direct Insert in GPH. Otherwise, the bucket overflow occurs in the Hash Table Bucket. It is essential to resolve the bucket overflow when it occurs as GPH guarantees exactly 1 probe for each lookup. In particular, we devise the Extended Move in GPH to handle bucket overflow during insert processing.

The general idea of Extended Move is to relocate the key-value pairs to other Hash Table Bucket which has vacant slots. In the perfect hashing scheme, it is not trivial as resolving the bucket overflow is changing the mapping from keys to buckets, which could affect all the mapped keys. To overcome it, Extended Move relocates the key-value pairs within R Hash Table Buckets to resolve the bucket overflow during insert processing.

Returning to the sketch of GPH in Figure 5, Extended Move first attempts to change Offset Δ_i to Δ_i', which determines the Hash Table Bucket $B[V_i^{j'}]$ via the hash function $\mathcal{H}_b$. The key-value pairs with Cell id i will be relocated in an effort to resolve the bucket overflow. Extended Move enumerates all possible Δ_i' to resolve the bucket overflow at Offset level as its value range is $[0, R-1]$. However, it is still possible the bucket overflow cannot be resolved at the Offset level. Thus, Extended Move changes the value of Mapping Φ_i, which first determines the new Virtual Bucket id j via the hash function $\mathcal{H}_v$, then identifies the new Hash Table Bucket $B[V_i^j]$ via the hash function $\mathcal{H}_b$. We illustrate the bucket overflow resolving procedure in Extended Move via the example in Figure 8, as the number of Virtual Buckets in a cell is $R = 4$. Bucket overflow occurs when inserting $(69, 2)$ into Hash Table Bucket 3 in GPH.

- **First, it attempts to change Offset.** Since $i = 4$ and $j = 2$ for $k = 69$, insert$(69, 2)$ probes $B[V_4^2 = 3]$, but the bucket 3 is full and bucket overflow occurs. Thus, Extended Move attempts to change Offset $\Delta_4 = 0$ to $\Delta_4' = 1$.
- **Second, it relocates key-value pairs with changed Offset.** The key-value pair $(69, 2)$ is inserted into Hash Table Bucket with id 5 via the new Offset $\Delta_4' = 1$. To guarantee the correctness, all the key-valued pairs inserted with the Offset $\Delta_4 = 0$ (all colored key-value pairs in Hash Table Buckets 0, 3, 6 in ①) will be relocated with the new Offset $\Delta_4' = 1$. See the colored key-value pairs in Hash Table Buckets 3, 5, 6 in ②). The key-value pair $(1, 4)$ causes bucket overflow on Hash Table Bucket 6 again. Extended Move then enumerates other possible Δ_4', e.g. 2 and 3 as $R = 4$ and 0, 1 has been enumerated. It cannot resolve the bucket overflow by changing Δ_4' in this example.
- **Third, it attempts to change Mapping Φ_i and relocates the key-value pairs.** Extended Move next attempts to change Mapping $\Phi_4 = 0$ to $\Phi_4{}' = 1$. It changes Virtual Bucket id j for all key-value pairs that were inserted with Φ_4 in $CELL[4]$, e.g., key-value pair $(48, 6)$ was originally inserted into $B[V_4^0 = 0]$, but now it should be inserted into $B[V_4^{2'} = 4]$. In this example, it is valid to have $\Delta_4{}' = 0$ and $\Phi_4{}' = 1$ to insert $(69, 2)$ in GPH, and Extended Move resolves bucket overflow and terminates.

To optimize the performance of insert kernel in GPH, we reduce unnecessary memory access by pruning invalid changes of Mapping and Offset before relocation, e.g., we skip the relocation that must overflow (e.g., the number of key-value pairs in a bucket exceeds M) or multiple Virtual Buckets of the same cell are relocated to the same Hash Table Bucket (i.e., $V_i^{j_1} = V_i^{j_2}$).

6.2 Theoretical Guarantee of Insert Kernel

Theorem 6.1 provides the theoretical guarantee to insert an arbitrary key-value pair into GPH.

Theorem 6.1. *The expected relocation times of key-value pairs within R Hash Table Buckets to insert an arbitrary key-value pair (k, v) to GPH is $O(1/(1-\alpha)^R)$, where R is the number of Virtual Buckets of a cell, and α is the load factor of GPH before (k, v) insertion.*

For the sake of presentation, let us prove the above theorem in a GPH, where each Hash Table Bucket has only one slot (i.e., $M = 1$). It is the upper bound of the relocation times of GPH with $M > 1$ as the key-value pairs mapped from different cells could co-exist in the same Hash Table Bucket in GPH with $M > 1$.

The number of key-value pairs stored in GPH is $d = \alpha \cdot N$. Now we Insert(k, v) on GPH. If it is a Direct Insert (i.e., writing to vacant slot or overwriting an existing key), there is no overflow. If there is an overflow, we assume that there exists a valid relocation. Then, the Extended Move attempts to change the Φ_i to insert (k, v) and resolves bucket overflow. We consider the worst case to relocate, i.e., all R Hash Table Buckets $B[V_i^0], B[V_i^1], \ldots, B[V_i^{R-1}]$ are occupied and should be relocated to R vacant buckets without overflow. Let X be the event of a successful relocation and T be the number of relocation times until the first successful X happens. In each relocation, the R key-value pairs that were inserted via $CELL[i]$ and are stored in $B[V_i^0], \ldots, B[V_i^{R-1}]$ are randomly relocated to R buckets in a total of N buckets with d full buckets. Thus, the probability of X is $Pr(X) = \binom{N-d}{R} / \binom{N}{R}$. Then, $E(T) = \sum_{l=1}^{\infty} l \cdot Pr(X)(1 - Pr(X))^{l-1} = 1/Pr(X)$ as each attempt is independent and follows a geometric distribution. Hence:

$$E(T) = \frac{\binom{N}{R}}{\binom{N-d}{R}} = \prod_{0 \le j < R} (1 + \frac{d}{N-d-j}) \le (1 + \frac{d}{N-d-R})^R \tag{2}$$

Since $N = \frac{d}{\alpha}$, $d \gg R$, and $0 \le \alpha \le 1$, we have

$$E(T) \le (1 + \frac{d}{N-d-R})^R = (1 + \frac{\alpha \cdot d}{d - \alpha \cdot (d+R)})^R \approx (\frac{1}{1-\alpha})^R \tag{3}$$

At this point, the proof is complete.

Discussion of Insertion Failure. Insertion failure occurs when the number of key-value pairs, which mapped to the buckets of a cell, exceeds $R \cdot M$. In other words, a valid relocation for the newly inserted key-value pair does not exist. We next analyze the probability of the insertion failure in GPH. Without loss of generality, we assume the keys are independent distributed, and the mapping from key to bucket is uniform, i.e., each key randomly selects one of N buckets with the probability $\frac{1}{N}$. For a given cell with R distinct buckets, the event that a key maps to any of these R buckets follows a Bernoulli trial with the success probability $\frac{R}{N}$. After d insertions, the number of keys mapped to the buckets of a cell follows the binomial distribution $B(d, \frac{R}{N})$ [48]. Let Y be the event that after d insertions, the number of key-value pairs mapped to the buckets of a given cell does not exceed $R \cdot M$. Thus, the probability of success insertion is $Pr(Y) = \sum_{k=0}^{R \cdot M} \binom{d}{k} \left(\frac{R}{N}\right)^k \left(1 - \frac{R}{N}\right)^{d-k}$. Hence, the probability of failure insertion is $1 - Pr(Y)$, which is smaller than 0.5% when $d \ll M \cdot N$, i.e., the probability is practically negligible, as it falls below the conventional statistical significance threshold [19]. For example, in our default experimental setting (see Section 7) with a 1.5GB table size and 70% load factor, the theoretical probability of insertion failure on a cell is only 0.00539%. Thus, the insertion failures on GPH are negligible.

6.3 Designs to Support Parallel Insert

With multiple Insert Groups performing insertions in parallel, it is challenging to resolve the race condition among these groups. Besides, it is critical to reduce the I/O cost caused by the data relocation. In this section, we introduce our designs to support parallel insert on GPU to address these challenges.

Efficient Direct Insert. When multiple h are processing different insert operations simultaneously, they could overwrite each other and read outdated data if they manipulate the same cell or bucket. Specifically, Direct Insert may be affected by race conditions, i.e., (i) after reading the cell and before writing to the Hash Table Bucket, the cell is modified by others; and (ii) after finding the vacant slot and before writing the key-value pair, the vacant slot is written by others. To overcome the race conditions, we split the insert kernel into two phases: Saturation Fill Phase, and Refinement Phase.

The Saturation Fill Phase only allows Direct Insert, and the insert operations that encounter bucket overflow will be left for the Refinement Phase. Therefore, there is no modification to the Cell Index to trigger the first race condition, and all insert operations in this phase are lock-free. The second race condition can be resolved by using atomic compare and swap (atomicCAS) to write key-value pair to slot, which ensures that the slot is vacant and avoids overwriting. In the Refinement Phase, we assign locks for each cell and Hash Table Bucket to ensure the correctness of reading from and writing to them.

Efficient Extended Move. We briefly summarize the major steps of supporting multiple parallel Extended Move on GPU.

- Request locks of $CELL[i]$ and R Hash Table Buckets associated with $CELL[i]$. A potential deadlock scenario may arise when each of two Insert Groups holds a lock of a Hash Table

Bucket that the other requires, resulting in both Insert Groups unable to proceed. We employ the wait-die scheme [52] to avoid deadlock by assigning lock access priority to each Insert Group according to their starting time.

- Employ Workspace in shared memory. Workspace is used to cache Hash Table Buckets for later data relocation and the metadata information for later use in validating Δ_i and Φ_i, e.g., the number of vacant slots and the number of key-value pairs inserted via $CELL[i]$ in a bucket.
- Attempt to modify Δ_i and check its validity based on the metadata of the cached buckets. The step does not require a global memory request because threads use the information from Workspace in shared memory to validate the modification.
- If no valid Δ_i is found, modify Φ_i and return to the first step of requesting additional locks for new mapped R Hash Table Buckets.
- If the valid Δ_i and Φ_i are found, we conduct data relocation in Workspace, then push the updated Hash Table Buckets from Workspace to global memory. Finally, release the locks.

7 Experimental Evaluation

We verify the efficiency of GPH via overall performance evaluation, case study and effectiveness study in this section.

7.1 Experimental Setting

Datasets. We used both synthetic and real-world datasets in the experimental evaluation. In particular, the synthetic datasets are Random, Lineitem, and TaoBenchmark, and the real-world dataset is Reddit. We follow existing work [40, 46] to remove the duplicate pairs in each dataset and use 4-byte integer keys and 4-byte integer values. The descriptions of each dataset are as follows.

- Random [6], applies the Fisher-Yates shuffle to the space of 2^{32} unsigned integers, then generates 100,000,000 key-value pairs.
- Lineitem [1], has 100,784,453 key-value pairs, which are generated by TPC-H. We hash the combination of three columns (`ORDERKEY`, `PARTKEY`, and `LINENUMBER`) to obtain the hashed keys in the Lineitem dataset, and hash `LINESTATUS` to obtain the corresponding values.
- TaoBenchmark [26], simulates the workload of TAO. In particular, TAO [23] is a distributed data store, which is designed for social graph workloads processing at Facebook. In this work, the dataset generated by TaoBenchmark contains 165,000,000 key-value pairs. Its lookup workload is highly skewed, i.e., the most frequent 20% of keys contribute 82.14% of lookup operations, as it is generated based on *assoc_get* requests.
- Reddit [4], is a real-world workload, which collects Reddit comment actions from Kaggle in May 2015. It has 54,160,677 key-value pairs. We hash `comment ID` as key and `username` as value by following the setting in DyCuckoo [46].

We generate the lookup workload for each dataset, which consists of 200,000,000 lookup keys. The lookup keys are uniformly distributed except TaoBenchmark.

Experimental Environment. All experiments are conducted on an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz server with an NVIDIA Tesla V100S GPU. All reported measurements are averaged by running the experiments three times. Following the settings of existing work [11, 39, 46], we also store the whole dataset on global memory and use cudaEvent to measure GPU kernel time.

Competitors and Configurations. We compare GPH with three representative GPU-based hash tables: CUDPP [11], DyCuckoo [46], and WarpCore [40]. All the competitors use their code

and default configurations. We use 32,768 thread blocks for lookup operations in GPH, and each threads block has 512 threads. The Lookup Group size $|g|$ and Insert Group size $|h|$ are 4 and 8 in GPH, respectively. The number of slots in a Hash Table Bucket M is 16. To address GPU shared memory constraints, the Cell Index in GPH is partitioned per-SM during device setup, with each SM assigned 98,304 1-byte cells in 96KB shared memory, and lookups resolve their target SM through a precomputed mapping finalized prior to kernel execution. Each cell is associated with 8 Virtual Buckets (R). The hash function maps multiple keys to an integer in GPH is defined as $\mathcal{H}_i(k_1, \ldots, k_n) = \mathcal{H}_{i,0}(\mathcal{H}_{i,1}(k_1) + \cdots + \mathcal{H}_{i,n}(k_n))$, and these hash functions are fmix32 from MurmurHash3 [13] with different seeds. We use C++ and CUDA to implement GPH. All compared GPU-based hash tables are compiled with nvcc of CUDA version 12.5 and compute capability 7.0.

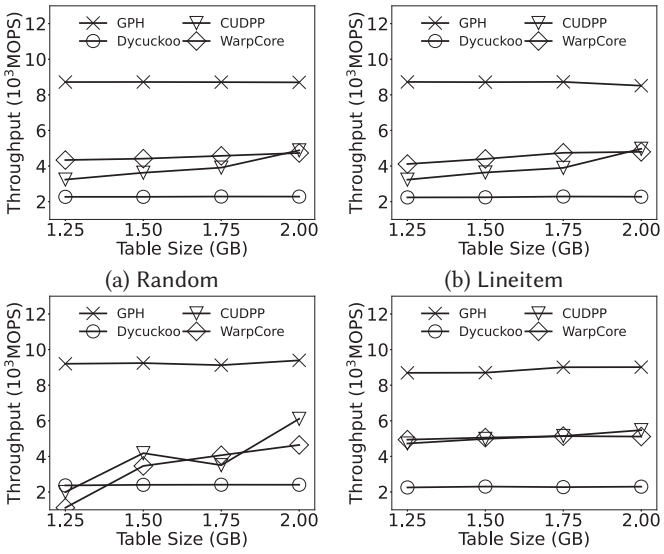
Parameters Tuning in GPH. The values of the parameters in GPH (e.g., M, $|g|$, $|h|$, R) affect its performance on different GPU devices. It is not trivial to determine the optimal parameter settings in GPH for different GPU devices and workloads. In GPH, the parameters can be set by benchmarking across hundreds of randomly selected combinations and selecting the best of them. We leave more efficient auto parameter setting on GPH as future work.

7.2 Performance Evaluation

In this section, we compare the lookup performance of GPH with competitors on different datasets and workloads. For every GPU-based hash table, we first construct its hash table on GPU global memory, then process the lookup operations in the workload.

Lookup Throughput by Varying Hash Table Size. We first evaluate the lookup throughput of the competitors and our GPH by varying table size from 1.25GB to 2GB. Each workload consists of 50% positive lookup (i.e., the lookup keys in the hash table) and 50% negative lookup (i.e., the keys are not in the hash table). Figure 9 depicts the experimental results among the four tested datasets. There is no doubt that our proposed GPH outperforms all competitors. It is the only GPU-based hash table that achieves over 8000 MOPS in all tested cases. In particular, the average improvement of GPH, on the four datasets, over the competitors CUDPP, DyCuckoo, and WarpCore, is 1.74-2.29×, 3.82-3.88×, and 1.75-2.78×, respectively. It confirms the superiority of GPH in lookup operations processing. In addition, we observe that all three competitors have a higher lookup throughput with a larger table size. The reason is that there are more vacant slots in a larger table so that their lookup operations can be terminated earlier when probing the vacant slots. Overall, GPH exhibits stable and superior lookup performance across different datasets and table sizes.

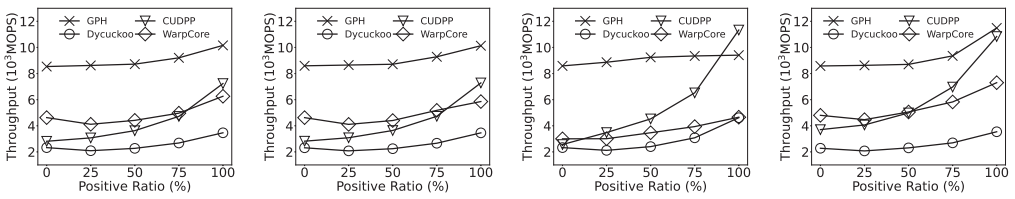
Lookup Throughput by Varying Positive Ratio. We next evaluate the throughput of lookup kernels by varying the positive ratio of lookup operations in the tested workloads. In these experiments, we set the hash table size as 1.5GB for all tested GPU-based hash tables, and the tested positive ratio ranges from 0% to 100% with a step size 25%. On the tested four datasets, as shown in Figures 10 (a)-(d), the improvement of GPH can be up to 2.11×, 3.63× and 2.52× over the existing GPU-based hash tables CUDPP, DyCuckoo, and WarpCore, respectively. The lookup throughput of all GPU-based hash tables (including our GPH) increases with the rise of positive ratio in the workload. The reason is that, compared to a positive lookup, a negative lookup typically needs more probing cost to determine the lookup key is not in the hash table. To optimize it, WarpCore determines the absence of the key as it encounters vacant slots. CUDPP employs an insert routine to enable an early-break strategy for negative lookup. Thus, the throughput of WarpCore and CUDPP on negative lookup outperforms that of DyCuckoo which requires two bucket probes for every negative lookup. However, the overall throughput of competitors on negative lookup still has enough space for improvement. For instance, none of the existing GPU-based hash tables reaches 5000 MOPS for the negative lookup workloads, i.e., the positive ratio of the workload is



(a) Random (b) Lineitem

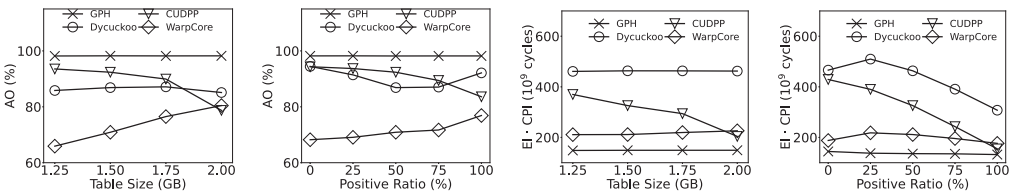
(c) TaoBenchmark (d) Reddit

Fig. 9. Lookup throughput, varying hash table size



(a) Random (b) Lineitem (c) TaoBenchmark (d) Reddit

Fig. 10. Lookup throughput by varying positive ratio

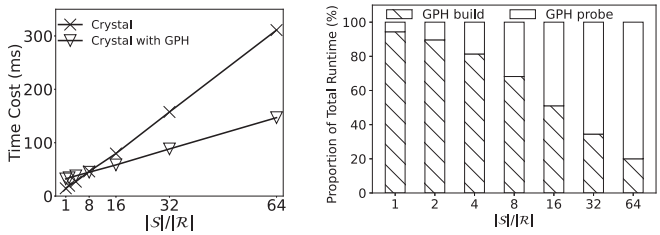


(a) *AO*, varying table size (b) *AO*, varying positive ratio (c) $EI \cdot CPI$, varying table size (d) $EI \cdot CPI$, varying positive ratio

Fig. 11. Performance analysis model evaluation

0%. However, the throughput of GPH for negative lookup workloads is over 8500 MOPS for all tested datasets. Interestingly, CUDPP is slightly better than GPH when the positive ratio in the workload is 100% on TaoBenchmark. The reason is that the lookup keys in TaoBenchmark are skewed and CUDPP caches the hot key-value pairs as its single-slot hash table entry occupies less cache memory.

Evaluation of Lookup Performance Analysis Model. In this experiment, we evaluate our proposed lookup performance model (i.e., the time cost is negatively correlated with *AO* and correlated with $EI \cdot CPI$), see Equation (1) in Section 3, on Random. We omit the experimental results on other datasets as they share similar observations and conclusions. In particular, we collect the measured metrics *AO*, *EI·CPI* by varying the data size and positive ratio.



(a) Comparison with Crystal (b) Breakdown of Crystal with GPH

Fig. 12. End-to-end GPU time cost of Crystal hash join

Figures 11(a) and (b) show the *AO* of all these GPU-based hash tables in all tested cases. The *AO* of GPH is the largest, which reaches almost 100%. It confirms (i) the Takeaway of Observation I in Section 3, and (ii) the effectiveness of the designs in GPH (i.e., guaranteeing exactly 1 probe for each lookup operation). For DyCuckoo, its *AO* is around 86% for many tested cases, and it has a low *AO* with 50% and 75% of positive ratio. The reason is that the number of probes of DyCuckoo is quite stable for full positive or full negative lookup operations, but the mixed positive and negative lookup operations increase the degree of inconsistency. CUDPP exhibits a relatively high *AO* but falls down with the rise of table size and positive ratio. The reason is that its number of probes for positive lookup is in a range of 1 to 4 entries, while most negative lookup operations have a consistent number of probes. WarpCore shows the lowest *AO* because of its open addressing scheme. Thus, the number of probes falls into a large range as some warps are unfairly assigned with lookup operations that need to probe much more entries than others.

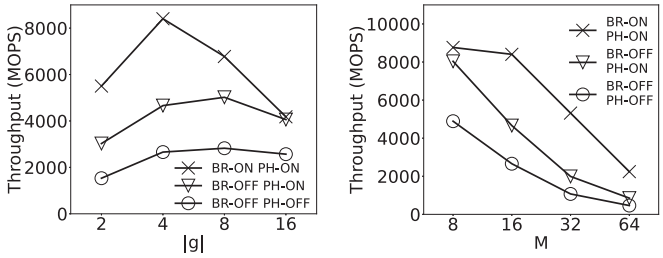
Figures 11(c) and (d) show that the GPH achieves stable 140 billion cycles $EI \cdot CPI$ on different table size and positive ratio, which is the smallest among evaluated competitors. The reason is that GPH chooses to use fewer threads in a cooperative group, thus, it has small a *EI*. In addition, the use of Bucket Requester in GPH results in small *CPI*. Interestingly, CUDPP and DyCuckoo have large $EI \cdot CPI$, especially for small table size and negative lookup. The reason is that small tables and negative lookups typically require more probes to hash tables, as small tables have few vacant slots and negative lookups need more probes to determine the result.

7.3 Hash Join Case Study on a GPU Database

In this section, we investigate the superiority of GPH in the hash join case study on a GPU database. Specifically, we replace the non-partitioned global hash table used in the hash join algorithm of the GPU database Crystal [51] with our GPH and verify its practical performance on hash join.

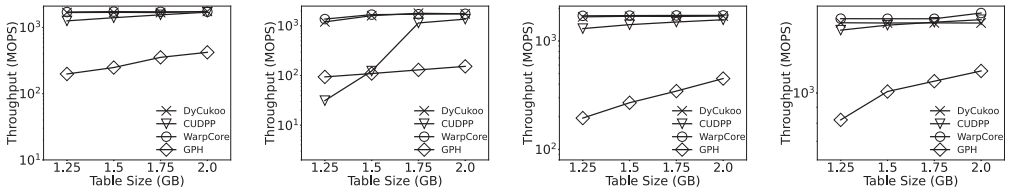
The hash join algorithm in Crystal builds a GPU-based open address hash table on the smaller relation $\mathcal{R}$ and probes it by using the larger relation $\mathcal{S}$. Both relations $\mathcal{R}$ and $\mathcal{S}$ are generated via a data generator for hash join studies [17, 28], the packed key-value pairs in $\mathcal{R}$ and $\mathcal{S}$ are 8 bytes. To evaluate the mixed lookup and insert workloads, we vary the size of relation $\mathcal{S}$ while keeping $\mathcal{R}$ fixed. In particular, the cardinality of $\mathcal{R}$ is 16,777,216, and the cardinality of $\mathcal{S}$ is correspondingly scaled to be $4 \cdot |\mathcal{R}|$, $8 \cdot |\mathcal{R}|$, $16 \cdot |\mathcal{R}|$, $32 \cdot |\mathcal{R}|$ and $64 \cdot |\mathcal{R}|$, which follows the setting in [51]. The load factor of the open address hash table in Crystal and our GPH are 50%, which is common for hash join workload [33, 41].

Figure 12(a) shows the time cost of Crystal and Crystal with GPH. The performance gain of GPH over the original hash table in Crystal becomes larger with the rising of probe relation size as GPH is optimized for accelerating lookup operations. The cut-off point between lookup and insert occurs when there are roughly 8 times more lookups than insertions. Figure 12(b) provides the breakdown of the cost of GPH. It illustrates that the proportion of the building cost varies from 94% to 20% when $|\mathcal{S}|$ grows from $1 \cdot |\mathcal{R}|$ to $64 \cdot |\mathcal{R}|$.



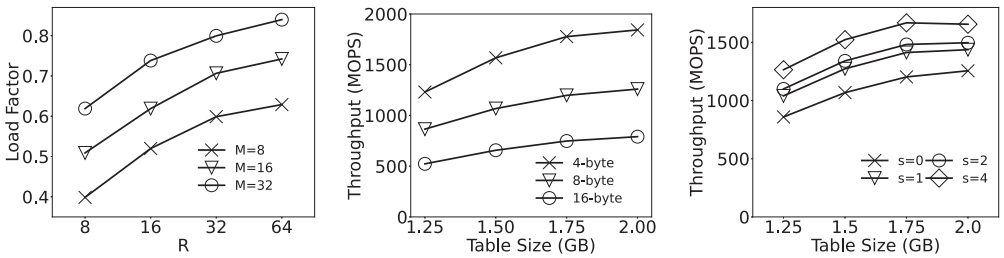
(a) varying $|g|$ (b) varying M

Fig. 13. Lookup throughput w.r.t. different settings



(a) Random (b) Lineitem (c) TaoBenchmark (d) Reddit

Fig. 14. Insert throughput, varying hash table size



(a) R and M (b) Key-value size (c) Skewness

Fig. 15. Effectiveness study of insert kernel on load factor and throughput

7.4 Effectiveness Study

Effect of the Designs in Lookup kernel. We first verify the effectiveness of the Bucket Requester module and perfect hashing design of GPH on Random. In particular, BR-ON PH-ON is our GPH, which enables Bucket Requester module and with prefect hashing scheme. BR-OFF PH-ON replaces the Bucket Requester module in GPH by the bucket request approach in [46]. BR-OFF, PH-OFF uses the bucket request approach and non-perfect hash scheme in [46]. We vary the number of threads in a Lookup Group $|g|$ and the number of slots in a Hash Table Bucket M. The results in Figure 13 depict that BR-ON, PH-ON outperforms BR-OFF PH-ON, and BR-OFF PH-ON outperforms BR-OFF PH-OFF in all cases. The overall performance gain of our GPH (i.e., BR-ON PH-ON) are from twofold: (i) GPH is a perfect hash table, which guarantees exactly 1 probe for each lookup; (ii) the lookup performance of GPH is further improved by Bucket Requester, which is built upon the perfect hashing scheme of GPH and improves $EI \cdot CPI$.

In addition, according to the comparison between BR-ON PH-ON and BR-OFF PH-ON, we observe that the Bucket Requester is most effective when $|g| = 4$, as the design of the Bucket Requester enhances each thread to access several slots. Interestingly, the performance of $|g| = 2$ is low. The reason is that the SM does not have enough registers to cache the vectors in this extreme case, resulting in low AO. Moreover, Bucket Requester enables similar performance for $M = 8$ and $M = 16$. Thus, we use 16 slots each bucket to improve the space efficiency.

Effect of the Designs in Insert Kernel. We last evaluate our design of insert kernel in all tested workload. GPH employs the perfect hashing scheme to accelerate the lookup performance. The trade-off in the design is that the insert procedure has to spend more time on data relocation and lock contention. As shown in Figure 14, the throughput of the insert kernel of GPH is worse than those of other GPU-based hash tables. However, it is still worth mentioning that GPH is the first GPU-based hash table with perfect hashing scheme that supports dynamic insert operation. The insert throughput of GPH is over 100 MOPS.

In Figure 15(a), we report the load factor by inserting key-value pairs into GPH with different numbers of associated Virtual Buckets in each cell (R) and different number of slots in each bucket (M). The results show that the load factor of GPH can be affected by varying R and M. These results provide some guidelines for users to specify the parameters (e.g., R and M) in GPH.

We also study the effect of varying the key and value sizes on insertion performance. Figure 15(b) illustrates the insertion performance for compacted key-value pair sizes of 4, 8, and 16 bytes. Compared to the default 8-byte size, the 4-byte size achieves an average throughput increase of 46.8%, and the 16-byte size results in an average throughput decrease of 38.4%. The results indicate that larger key-value sizes lead to lower insert throughput, as they consume more bandwidth.

We last investigate the impact of skewness on insertion performance in Figure 15(c). The skewness is represented by 50,000,000 data records generated from a Zipf distribution, where the parameter s ranges from 0 (indicating uniform distribution) to 4 (indicating highly duplicated keys). The results indicate that GPH demonstrates improved insertion performance with skewed data. The reason is that inserting duplicate keys triggers the overwrite (similar to previous GPU-based hash tables [14, 46, 55]) in Direct Insert so it avoids the expensive overflow handling Extended Move.

8 Conclusion

In this work, we propose an effective-and-generic lookup performance model with insightful observations that reveal the performance bottleneck in existing GPU-based hash tables. Then, we propose a novel GPU-based hash table GPH to accelerate the lookup performance. We employ the perfect hashing in GPH. To further improve the lookup throughput, the Bucket Requester module is proposed, which exploits vectorization and instruction-level parallelism techniques. We also devise the insert kernel to support parallel insertions in GPH. To our best, it is the first GPU perfect hashing with dynamic insert operation supports. We demonstrate the superiority of GPH by extensive experiments on both synthetic and real-world datasets. In the future, GPH will be enhanced by two-fold: (i) devising auto parameter setting method for GPH, and (ii) supporting other data types (e.g., strings, objects) on GPH to improve its usability in different workloads.

Acknowledgments

We are grateful to the anonymous reviewers and the shepherd for their insightful comments and valuable suggestions. Bo Tang was supported by National Science Foundation of China (NSFC No. 62422206) and a research gift from AlayaDB.AI. Man Lung Yiu was supported by Hong Kong Research Grants Council (GRF 152043/23E). Jianbin Qin was supported by National Natural Science Foundation of China (NSFC No. 62472289), Shenzhen Natural Sponsored by Science Foundation (20220810142731001), The Pearl River Talent Recruitment Program (2019ZT08X603), and Guangdong Province Key Laboratory of Popular High Performance Computers (2017B030314073). We also thank Qihao Ye and Yufan Xiang for the discussions at the early stage of this work.

References

[1] 2011. TPC-H DBGEN. https://github.com/electrum/tpch-dbgen

[2] 2013. CUDA pro tip: Increase performance with vectorized memory access. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/
[3] 2015. Achieved Occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.
[4] 2019. May 2015 Reddit Comments. https://www.kaggle.com/datasets/kaggle/reddit-comments-may-2015.
[5] 2024. CUDA C++ Programming Guide Performance Guidelines. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[6] 2024. CUDA Data Parallel Primitives library. https://github.com/cudpp/cudpp/blob/master/apps/cudpp_hash_testrig/random_numbers.cpp
[7] 2024. Kernel Profiling Guide: Sections and Rules. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#id12.
[8] 2024. Kernel Profiling Guide: Source Metrics. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#id34.
[9] 2024. NSIGHT COMPUTE. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.
[10] 2024. REDIS. https://redis.io/.
[11] Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2009. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia*. 1–9.
[12] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2012. Building an efficient hash table on the GPU. In *GPU Computing Gems Jade Edition*. 39–53.
[13] Austin Appleby. 2024. MURMURHASH3. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.
[14] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *IPDPS*. 419–429.
[15] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS*. 53–64.
[16] Muhammad A Awad, Saman Ashkiani, Serban D Porumbescu, Martín Farach-Colton, and John D Owens. 2023. Analyzing and implementing GPU hash tables. In *APOCS*. 33–50.
[17] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, main-memory joins: sort vs. hash revisited. *PVLDB* 7, 1 (2013), 85–96.
[18] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *ESA*. 682–693.
[19] Daniel J Benjamin, James O Berger, Magnus Johannesson, Brian A Nosek, E-J Wagenmakers, Richard Berk, Kenneth A Bollen, Björn Brembs, Lawrence Brown, Colin Camerer, et al. 2018. Redefine statistical significance. *Nature human behaviour* 2, 1 (2018), 6–10.
[20] Rajesh Bordawekar. 2014. Evaluation of parallel hashing techniques. In *Nvidia GTC*.
[21] Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and space-efficient minimal perfect hash functions. In *WADS*. 139–150.
[22] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *USENIX ATC*. 281–294.
[23] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*. 49–60.
[24] Qi Chen, Hao Hu, Cai Deng, Dingbang Liu, Shiyi Li, Bo Tang, Ting Yao, and Wen Xia. 2023. EEPH: An Efficient Extendible Perfect Hashing for Hybrid PMem-DRAM. In *ICDE*. 1366–1378.
[25] S. Chen, A. Ailamaki, P.B. Gibbons, and T.C. Mowry. 2004. Improving hash join performance through prefetching. In *ICDE*. 116–127.
[26] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, et al. 2022. Taobench: An end-to-end benchmark for social network workloads. *PVLDB* 15, 9 (2022), 1965–1977.
[27] Zbigniew J Czech, George Havas, and Bohdan S Majewski. 1992. An optimal algorithm for generating minimal perfect hash functions. *Information processing letters* 43, 5 (1992), 257–264.
[28] Yangshen Deng, Shiwen Chen, Zhaoyang Hong, and Bo Tang. 2024. How Does Software Prefetching Work on GPU Query Processing?. In *DaMoN*. Article 5, 9 pages.
[29] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.* 23, 4 (1994), 738–761.
[30] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *NSDI*. 371–384.
[31] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[32] Michael L Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a sparse table with 0(1) worst case access time. *J. ACM* 31, 3 (1984), 538–544.
[33] Oded Green. 2021. HashGraph—Scalable Hash Tables Using a Sparse Graph Data Structure. *ACM Trans. Parallel Comput.* 8, 2, Article 11 (July 2021), 17 pages.
[34] Tobias Groth, Sven Groppe, Thilo Pionteck, Franz Valdiek, and Martin Koppehel. 2022. Accelerated parallel hybrid GPU/CPU hash table queries with string keys. In *DEXA*. 191–203.
[35] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *SIGMOD*. 511–524.
[36] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *PVLDB* 16, 13 (2023), 4268–4281.
[37] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. 2024. PHOBIC: Perfect Hashing With Optimized Bucket Sizes and Interleaved Coding. In *ESA*, Vol. 308. 69:1–69:17.
[38] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*. 152–163.
[39] Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2018. WarpDrive: Massively parallel hashing on multi-GPU nodes. In *IPDPS*. 441–450.
[40] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. WarpCore: A Library for fast Hash Tables on GPUs. In *HiPC*. 11–20.
[41] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *DaMoN*. 55–62.
[42] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Stadium hashing: Scalable and flexible hashing on gpus. In *PACT*. 63–74.
[43] Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect spatial hashing. *ACM Transactions on Graphics* 25, 3 (2006), 579–588.
[44] Brenton Lessley and Hank Childs. 2019. Data-parallel hashing techniques for GPU architectures. *TPDS* 31, 1 (2019), 237–250.
[45] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*. 1–14.
[46] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. DyCuckoo: dynamic hash tables on gpus. In *ICDE*. 744–755.
[47] Prabhakar Misra and Mainak Chaudhuri. 2012. Performance evaluation of concurrent lock-free data structures on GPUs. In *ICPDS*. 53–60.
[48] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis* (2nd ed.). Cambridge university press. 92–93 pages.
[49] David S Moore and George P McCabe. 1989. *Introduction to the practice of statistics*. WH Freeman/Times Books/Henry Holt & Co.
[50] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.
[51] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *SIGMOD*. 1617–1632.
[52] Abraham Silberschatz, Greg Gagne, and Peter Baer Galbin. 2008. *Operating System Concepts* (8th ed.). Wiley. 741–742 pages.
[53] Vasily Volkov. 2010. Better performance at lower occupancy. In *Nvidia GTC*, Vol. 10. 16.
[54] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. 2021. Mapembed: Perfect hashing with high load factor and fast update. In *SIGKDD*. 1863–1872.
[55] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A case for gpus to maximize the throughput of in-memory key-value stores. *PVLDB* 8, 11 (2015), 1226–1237.
[56] Shijie Zhou and Viktor K. Prasanna. 2015. Scalable GPU-Accelerated IPv6 Lookup Using Hierarchical Perfect Hashing. In *GLOBECOM*. 1–6.

Received October 2024; revised January 2025; accepted February 2025