Designing GPU Data Structures for Efficient Memory Oversubscription

VIPIN PATEL*, Indian Institute of Technology Kanpur, India
SRINJOY SARKAR*, Indian Institute of Technology Kanpur, India
SWARNENDU BISWAS, Indian Institute of Technology Kanpur, India
MAINAK CHAUDHURI, Indian Institute of Technology Kanpur, India

Efficient concurrent data structures are important building blocks for accelerating applications on GPUs. With the ever-increasing memory footprint of GPU workloads, data structures used by kernels can exceed global memory capacity. Using the unified virtual memory (UVM) model is a popular approach for kernels to oversubscribe GPU memory without the need for explicit memory management by a programmer. However, we show that data structures executing with UVM can suffer from performance degradation due to the high overheads associated with data migration and thrashing for irregular access patterns.

In this paper, we propose two-level hierarchical designs for hash table and skip list data structures that aim to maximize access locality and handle use cases where the data structure oversubscribes GPU memory. The outer-level container enables efficient jumps to desired regions of the data structure, while the inner container allows operating on the data. The inner container is sized to facilitate efficient data transfers between the CPU and the GPU. Experimental results on a diverse set of input operation sequences show that our data structure designs substantially improve performance over optimized UVM baselines while supporting high degrees of GPU memory oversubscription. Importantly, our proposed design, when used to implement key-value stores in metagenomics classification and k-mer counting applications, achieves a geomean speedup of 2.06× for hash table and 2.37× for skip list over baseline UVM implementations.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; **Massively parallel algorithms**; **Graphics processors**; • **Theory of computation** → **Data structures design and analysis**.

Additional Key Words and Phrases: GPUs, concurrent data structures, hash tables, skip lists

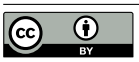
ACM Reference Format:
Vipin Patel, Srinjoy Sarkar, Swarnendu Biswas, and Mainak Chaudhuri. 2026. Designing GPU Data Structures for Efficient Memory Oversubscription. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 136 (April 2026), 30 pages. https://doi.org/10.1145/3798244

1 Introduction

The high compute capability and memory bandwidth of GPUs, coupled with energy efficiency, have spurred their usage across a diverse set of application domains including scientific computing simulations, big data and large-scale graph analytics [36], and computational biology [53]. Improved compute capabilities have led to a steady rise in the working set sizes of GPU-accelerated applications, which now often far exceed the GPU global memory capacity [15, 41]. Hence, the

*Both authors contributed equally to this research.

Authors' Contact Information: Vipin Patel, Indian Institute of Technology Kanpur, India, vipinpat@cse.iitk.ac.in; Srinjoy Sarkar, Indian Institute of Technology Kanpur, India, srinjoys23@cse.iitk.ac.in; Swarnendu Biswas, Indian Institute of Technology Kanpur, India, swarnendu@cse.iitk.ac.in; Mainak Chaudhuri, Indian Institute of Technology Kanpur, India, mainakc@cse.iitk.ac.in.



ACM 2475-1421/2026/4-ART136
https://doi.org/10.1145/3798244

traditional copy-then-execute (CTE) programming model is giving way to the newer unified virtual memory (UVM) paradigm, where the CPU virtual address space is also visible to GPU kernels. GPU vendors have introduced memory virtualization support through features such as unified virtual addressing, demand paging, and prefetching. UVM allows the host CPU and discrete GPU devices to share a unified virtual address space, enabling the GPU runtime to page memory in and out as needed [5, 15, 28, 29]. Demand paging in UVM-capable GPUs facilitates transparent memory oversubscription, improving programmer productivity, while the cost of servicing GPU page faults can be reduced by prefetching a chunk of nearby pages [21, 41, 75, 76].

Writing correct but scalable multithreaded GPU kernels is a difficult art [58, 71]. In addition to the algorithm and the compute capabilities of the GPU device, the scalability of the application depends on the data structures used. For example, the performance of the Pinterest application improved substantially by switching to GPU-accelerated hash tables [51]. Therefore, the design of more complex structures such as lists and trees is an active area of research in the domain of GPU computing [61]. Much recent work has focused on designing efficient data structures (e.g., hash tables, skip lists, B-trees, and membership query filters) for use in GPU code [6–9, 13, 14, 20, 22, 30, 31, 38, 38, 39, 44, 46, 47, 49, 52, 67, 72, 73, 77].

The Problem. The steady increase in the working set size and irregular access patterns of GPU applications require efficient memory management so that the GPU data structures efficiently scale with the input size. However, existing studies primarily focus on the traditional CTE model and do not address how to design efficient data structures that oversubscribe memory, and developers fall back to UVM. While UVM simplifies programming, its performance remains a significant concern due to frequent *far faults*[1], high address translation overheads, and high volume of page replacement [3, 5, 10, 15, 21, 26, 33, 41, 64, 70, 75]. For instance, Lin et al. [42] find UVM-based implementations competitive with the CTE model for workloads that fit in GPU memory, but about 7× slower with oversubscription. The performance degradation with UVM is more prominent in applications with irregular access patterns and pointer chasing [3–5].

Hash tables and skip lists are two popular concurrent data structures for storing key-value pairs [27, 38, 72]. Hash tables are widely used in various scientific applications like metagenomics sequencing [36], photodna [66], and database indexing workloads. Skip lists are popularly used in databases [24, 45, 63], networking [48], persistent memory [34, 40], concurrency control [12, 37, 74], and for representing skewed data access patterns [11, 17]. Such applications, when run on GPUs, often oversubscribe memory. For example, a genome database stored in a hash table is used for query processing in metagenomics sequencing. The size of this database can grow to 50+ GB, easily overflowing the GPU memory capacity.

Both hash tables and skip lists are characterized by input-dependent access locality [27]. Lack of locality in the input key strings can lead to irregular access patterns and poor data locality. When memory oversubscription is coupled with these performance pathologies, these data structures exhibit very poor scalability. In this paper, we establish that UVM-based implementations of hash tables and skip lists with oversubscription do not scale well with input size even when these implementations are optimized using data placement hints and prefetches based on memory/translation usage [50]. We observe that the run times of these implementations increase exponentially with input sizes indicating a clear *performance bottleneck*. We focus on alleviating this performance pathology by proposing efficient designs of concurrent hash tables and skip lists for GPUs.

[1]Far faults, which occur when a requested page is absent in GPU global memory, require multiple PCIe roundtrips, and are resolved after interacting with the CPU page tables. On NVIDIA GPUs, a far fault from a CUDA thread stalls the progress of the warp containing the thread, possibly affecting the overall GPU throughput.

Our Approach and Contributions. We propose novel designs of concurrent hash tables and skip lists with improved locality while supporting efficient oversubscription and smart management of pages resident in the GPU memory. Our design innovation involves a two-level anatomy of the hash table and the skip list. In our two-level hash table proposal, the outer structure is a hash table, where each slot points to an inner judiciously-sized hash table. We assume keys and values to be unsigned integers each of size 4 bytes. The set of possible keys (i.e., $[1, \texttt{UINT32\_MAX} - 1]$) is divided into M equal-sized ranges, and the size of the outer hash table is M.[2] If we denote the size of a range by R, all keys in the range $[i * R,\ ((i + 1) * R) - 1]$ map to the same slot of the outer hash table, and are stored in the corresponding inner hash table, thereby offering locality within the range of keys. The inner hash table is sized to be a small integral multiple of the GPU page size enabling efficient page management and migration. We further propose a processing pipeline which includes forming batches from the input data and reordering the input to improve data locality [44, 77].

Our skip list proposal uses the GPU-friendly skip list (GFSL) algorithm [47] as the baseline. First, we optimize the baseline GFSL to ameliorate scalability bottlenecks arising from lock contention. Next, we enhance it with UVM support by proposing a two-level hybrid structure where the outer container is a hash table, the slots of which point to smaller skip lists. Each inner-level skip list stores the keys of a range. In contrast to a monolithic skip list design like GFSL, the outer hash table in our design quickly narrows the search for a key to the desired sub-region of the skip list, and avoids generating a series of page faults for traversing the horizontal and vertical pointers in the skip list. We apply the same processing pipeline as in our hash table proposal to the skip list as well. This showcases the effectiveness and generality of our design paradigm across different data structures under memory oversubscription. Our designs, when used to implement key-value stores in metagenomics classification and k-mer counting applications [35, 36], achieve up to 2.3× and 4.8× speedup for hash table and up to 2.6× and 35× speedup for skip list over an optimized UVM-based baseline. This paper makes the following contributions:

- shows that UVM implementations of popular data structures do not scale well (Section 3),
- proposes a hierarchical design for hash table that improves data locality and allows for efficient page migration (Section 4.1),
- proposes performance optimizations on the state-of-the-art GFSL algorithm for skip list and extends the design to reduce page faults under oversubscription (Section 4.2),
- presents a thorough evaluation that highlights performance gains of our proposal compared to a meticulously optimized UVM-based baseline (Section 5), and
- highlights performance benefits of our proposal on metagenomics classification [36] and k-mer counting [35] applications (Section 6).

2 Existing GPU Data Structures

In this section, we briefly review the existing designs of hash tables and skip lists for GPUs.

2.1 Hash Tables

Hash tables are used for tracking key-value relationships in a sparse domain because of their expected constant time complexity for different operations like insert and search. The performance of a hash table depends on the memory layout, access pattern dictated by the sequence of keys in the input operation string, quality of the hash functions, collision resolution schemes, and dynamic resizing techniques. Different approaches have been explored for CPUs [27, 43] and many libraries provide efficient implementations with different features [18, 19].

[2]We reserve zero (called `SENTINEL_KEY`) to denote empty slots and `UINT32_MAX` (called `TOMBSTONE_KEY`) to denote a deleted key.

Hash tables for GPUs can be divided into static [2, 23, 25, 31, 32, 46] and dynamic [6, 8, 14, 30, 39, 77] designs. The container array is not resizable in static hash table designs, which implies that these designs rely on a worst-case estimate of the number of unique keys for allocating memory. Static allocation without an accurate estimate of the number of unique keys can waste memory.

GPU data structure kernels provide bulk APIs that work on multiple data elements in parallel to avoid frequent expensive kernel launches [6, 14, 31, 44, 47]. A natural division of work in bulk-API-based kernels is to assign an operation on a (key, value) pair to a CUDA thread. However, given the potentially irregular input-dependent access pattern, assigning a different key to each thread leads to control and memory divergence within a warp. The Warp-Cooperative Work-Sharing (WCWS) strategy [6, 30–32, 47], tries to ameliorate this problem by assigning an operation to a warp. With WCWS, all threads of a warp cooperatively complete the operation assigned to the warp.

SlabHash [6] uses closed addressing for conflict resolution. Unlike traditional linked lists, each bucket in SlabHash is organized as a list of super-nodes. Each super-node consists of multiple key-value pairs and a pointer to the next node, and is sized based on the warp size (e.g., 32 for NVIDIA GPUs). *DACHash* [77] extends SlabHash to sort the input keys for better access locality, and dynamically assigns work to either a thread or a warp based on the super-node count in the relevant bucket.

WarpCore [31] proposes a static hash table with open addressing for resolving collisions. WarpCore assigns an operation on a (key, value) pair to one cooperative group (CG) following the WCWS strategy, and all threads in a CG participate to carry out the operation (e.g., performs a parallel probe to resolve conflicts). WarpCore may suffer from high overheads when there is a large proportion of negative queries (e.g., failed delete or search operations) at high load factors because it has to keep iterating through the hash table buckets till it encounters an empty slot [8].

DyCuckoo [39] divides the hash table array into d smaller tables of equal capacity and uses Cuckoo hashing [59] to lower the cost of rehashing when the table is resized. The hash table is downsized or upsized only if the load factor crosses respective thresholds.

GPH [14] improves performance by using a prefect hashing scheme. GPH divides the hash table into n contiguous buckets, and stores the location of each bucket in an auxiliary array in per-SM shared memory. GPH uses three hash functions to map a key k to a bucket. The first function maps k to an auxiliary array index I. The second function calculates a virtual bucket ID using k and the mapping stored at index I. The third function computes the destination bucket. The auxiliary array allows GPH to perform an operation with a fixed number of probes. However, a small fraction of insert operations may fail due to collision.

2.2 Skip Lists

A skip list [62, 72] is a probabilistic data structure that allows efficient insertion, deletion, and search operations on sorted linked lists. A skip list is organized as a hierarchy of linked lists, where higher-level lists are always contained in lower-level lists. The list at the lowest level (L_0) contains the data elements in a sorted order. We refer to L_0 as the data layer, and the upper levels as the index layers. Unlike balanced trees, a skip list does not require expensive rebalancing after mutation operations while achieving an expected logarithmic search time.

Misra and Chaudhuri [46] ported the linearizable construction of a lock-free skip list algorithm described by Herlihy and Shavit [27] to GPUs. Moscovici et al. [47] proposed GPU-friendly skip list (GFSL) which tries to reduce the memory footprint and performance penalty due to uncoalesced global memory accesses on a GPU. GFSL also uses the concept of super-nodes to build the linked lists at each level. Figure 1 shows the organization of a super-node in the index and data layers of GFSL. Each super-node in GFSL contains $N + 2$ (key, value) pairs. A value field in a super-node of an index layer i stores the pointer (referred to as `DownPtr`) to the relevant super-node in the layer

$i - 1$. The same value field in a super-node of the data layer stores the data value associated with the corresponding key. The last two pairs of fields in a super-node are used to store the maximum key in the super-node (`MaxKey`), a pointer to the next super-node at the same level (`NextPtr`), and a lock to ensure mutual exclusion during concurrent mutations. Figure 2 shows a schematic of the GFSL design where each super-node accommodates two keys. The bold dotted line shows the path taken while searching for the key 11. A super-node is split if an insert operation finds the node to be full, and a new super-node is added to the skip list containing half of the values from the split parent super-node. In contrast to the traditional skip list, a key in GFSL is raised to the next higher level only on a super-node split operation. Like hash tables, the super-node is sized to match the size of a warp (i.e., $N + 2$ is 32).

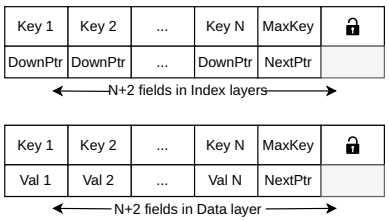


Fig. 1. Super-nodes in the index and data layers of GFSL

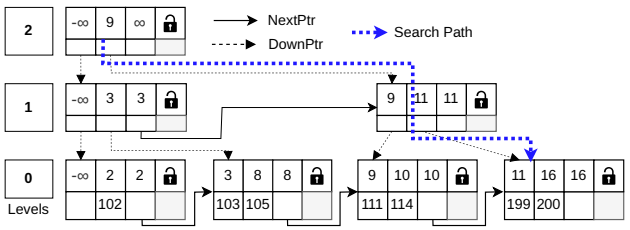


Fig. 2. GFSL with two keys per super-node. The bold dotted line shows the path traversed while searching for the key 11.

While GFSL is a lock-based design for discrete-memory GPUs, CMSL [20] is a lock-free skip list designed for Intel's integrated GPUs using the "C for media" framework. CMSL has 16 keys per super-node, and also employs instruction-level parallelism using vector instructions to read two super-nodes in parallel. CMSL relies on the SIMD16 CAS instruction exclusive to Intel GPUs for performing updates to the data layer blocks.

3 Designing UVM-based Data Structures

Existing hash table and skip list design proposals do not consider GPU memory oversubscription. As a first step toward enabling memory oversubscription in these data structures, we discuss UVM-based implementations of hash table and skip list that we use as the *baseline* for this work. We also point out the bottlenecks faced by these UVM-based data structure designs and why existing techniques that aim to optimize UVM performance fail to address them.

3.1 Hash Tables with UVM Support

Our hash table implementation extends WarpCore [31] to support the UVM model. Henceforth, we will refer to our UVM-enabled baseline hash table as HT-UVM. HT-UVM's hash table is static, similar to WarpCore. Hence, HT-UVM conservatively allocates an array of size $\lceil$`(NUM_INS/LF)`$\rceil$ in global memory using managed memory in CUDA, where `NUM_INS` is the number of insert operations to be performed on the hash table and `LF` indicates the fraction of filled entries. HT-UVM implements three kernels: `insert()`, `delete()`, and `search()`. The `insert()` kernel takes as input a pointer to the hash table in GPU global memory and an array of key-value pairs that is to be inserted. The `delete()` and `search()` kernels take a pointer to the hash table and an array of input keys as parameters. The `delete()` and `search()` kernels also pass an output array which tracks the values (corresponding to the input keys) that were operated on. The kernels in HT-UVM can be invoked both for a single key or for a batch of keys; the latter avoids the overhead of repeated kernel launches. HT-UVM

implements WCWS. Algorithm 1 shows the high-level steps in the `insert()` kernel. The kernel uses warp intrinsics to find an empty slot to insert the key and the value.

HT-UVM uses open addressing with double hashing for collision resolution. Figure 3 shows how collisions are detected and resolved. The figure shows a CG having four threads attempting to insert a key x with value y. Suppose the key x hashes to the slot denoted by $h_1(x)$, but the slot is already occupied by key K_i. The other threads in the CG parallelly probe the remaining slots after K_i to find an empty slot where the tuple (x, y) can be inserted. If all the slots probed by a CG are occupied (indicated by the shaded slots K_i to K_{i+3}), the CG probes alternate locations in the hash table (indicated by the slots K_j to K_{j+3}) depending on the secondary hash function $h_2(x)$. Threads with ranks 0 and 1 in the CG find slots K_j and K_{j+1} occupied, while the threads with ranks 2 and 3 find slots K_{j+2} and K_{j+3} unoccupied. The CG chooses the earliest available slot K_{j+2} as the index for inserting x, and the corresponding CG thread attempts to atomically store (x, y) in slot K_{j+2}. The insert operation completes on a successful atomic store. A failed attempt indicates a concurrent insert to K_{j+2}, and the CG reattempts insertion starting from the next empty slot K_{j+3}.

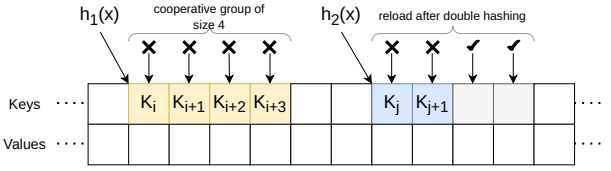


Fig. 3. Collision resolution in HT-UVM while inserting (x, y)

The probing scheme in HT-UVM differs from WarpCore in deciding when to terminate a search. In WarpCore, a CG terminates its probe sequence either upon encountering a `SENTINEL_KEY` or after retrying for a maximum of $\lceil \frac{(2^{64}-1)}{2^{CGS}} \rceil$ times to ensure complete scanning of the whole array, where *CGS* is the size of the CG. In contrast, our HT-UVM design ensures that the size of the CG is co-prime to the size of the hash table [1]. This guarantees that every slot in the hash table will be visited within a number of probes equal to the hash table size. This approach significantly reduces the overhead during negative queries, while still ensuring complete coverage of the probe space.

UVM Support. To enable a vanilla UVM support, we incorporate managed memory allocation for the hash table array, i.e., replace `cudaMalloc()` and `cudaMemcpy()` calls by `cudaMallocManaged()` calls. However, the vanilla implementation performs very poorly with more than 50% oversubscription. A general good-quality hash function distributes the memory accesses across different buckets, which leads to touching random pages depending on the input key pattern. Oversubscription further leads to many pages storing hash table data to get evicted (based on variants of least-recently-faulted replacement policy implemented by the contemporary NVIDIA GPUs [5]). This gives rise to frequent far faults. Far faults are expensive because they incur overheads of page unmapping on the CPU, page migration to the GPU, and installing page table entries in the GPU. Multiple concurrent warps (or CGs) stall waiting for their respective far faults to be resolved, thereby hurting performance [64, 65]. For example, in our experiments (details discussed in later sections), the `insert()` kernel does not finish within even 20 hours with 100% oversubscription[3] on a GPU with 16 GB memory.

We optimize data movement by incorporating memory advise and prefetching hints [54]. The best combination of hints for the `insert()` kernel found through empirical evaluation is `cudaMemAdviseSetAccessedBy` and `cudaMemPrefetchAsync` for the hash table array and the input and auxiliary arrays (not shown in Algorithm 1). The first hint indicates that the hash table

[3] 100% oversubscription means the application's memory footprint is 2× the size of GPU's global memory.

Algorithm 1: Insert Operation in HT-UVM

```
Input: Hash table HT, Key K, Value V
index ← (primaryHash(K)) mod getSize(HT)
if K = HT[index].key then                                   // Key exists, update value
    HT[index].value ← V
    return false
while ∃ unchecked slots ∈ HT do                             // Search for an empty slot
    while cooperative group cg finds at least one empty slot do                  // WCWS
        leader ← getFirstEmptySlot()
        if cg.thread_rank() == leader then          // Current thread found an empty slot
            old ← atomicCAS(&HT[index].key, SENTINEL_KEY, K)
            if old = SENTINEL_KEY then                      // Current thread inserted K
                HT[index].value ← V
                return true
        if a thread in cg found a key matching K then      // K ∈ HT[index ... index + cg.size()]
            HT[index].value ← V
            return false
    index ← (index + secondaryHash(K)) mod getSize(HT)          // Double hashing probe
return false
```

array, the input array, and the auxiliary array will be predominantly accessed by the GPU, allowing the runtime to pre-map the page table entries on the GPU without actually migrating the data. The `cudaMemPrefetchAsync` hint allows the runtime to asynchronously prefetch the hinted pages before the kernel starts, thereby reducing the initial far faults. We also experimented with the `cudaMemAdviseSetPreferredLocation` and `cudaMemAdviseSetReadMostly` hints. The `cudaMemAdviseSetPreferredLocation` hint indicates a preferred location (CPU or GPU) for a memory region, but it does not improve performance because the hash table pages anyway get evicted due to oversubscription. The `cudaMemAdviseSetReadMostly` hint is not applicable as we update the hash table across kernel launches.

3.2 Skip Lists with UVM Support

We extend GFSL [47] with UVM support to enable oversubscribing the GPU memory, and refer to this design as SL-UVM. SL-UVM supports `insert()`, `delete()`, and `search()` kernels, similar to HT-UVM. In the following, we first briefly describe the important operations of SL-UVM, based on the GFSL proposal. Next, we discuss how we enable a vanilla UVM support on top of GFSL.

Efficiently Locating the Candidate Node. Given an input key K, a common requirement across insert, delete, and search operations is to efficiently determine the location of a candidate super-node containing key K. Algorithm 2 shows the pseudocode for finding the candidate super-node. To determine the location of the candidate super-node, a warp starts traversal with the first super-node at the topmost index layer. The penultimate thread in the warp compares K to the `MaxKey` field of the super-node. If `MaxKey` is smaller than K, the search continues to the next super-node at the same index layer following the pointer `NextPtr`. On the other hand, if K is less than the `MaxKey`, the warp probes all data locations in the super-node simultaneously to identify the largest key $K' \leq K$. The pointer `DownPtr` corresponding to K' is used to traverse down to the next lower index layer. If all the keys in a super-node are greater than K, the algorithm backtracks to the previous node, and

Algorithm 2: Find Candidate Node in a Skip List

Input: Skip list SL, search key K
Output: Candidate super-node $Node$ at data layer L_0 such that $K \in Node$

```
1  L_top ← Highest non-empty index layer in SL;  level ← L_top
2  Node_curr ← First node in the index layer L_top;  Node_prev ← NULL
3  while level > 0 do                    // Traverse down the index layers to reach the data layer
4      while Node_curr.MaxKey < K do     // Lateral traversal in current index layer
5          Node_prev ← Node_curr;  Node_curr ← Node_curr.NextPtr
6      ProbeSuperNode(Node_curr, K)      // Parallel comparison using warp intrinsic
7      if ∀x ∈ Node_curr.Keys,  x > K then
8          Node_curr ← Node_prev.DownPtr[Node_prev.MaxKey]                    // Backtrack
9      else                              // Down traversal: find the largest key K' such that k < K
10         K' ← max{x ∈ Node_curr.Keys | x < K}
11         Node_curr ← Node_curr.DownPtr[K']                                  // Move down
12     level ← level − 1
   // Search through the data layer L_0
13 while Node_curr.NextPtr ≠ NULL ∧ K > Node_curr.MaxKey do
14     ProbeSuperNode(Node_curr, K)
15     Node_curr ← Node_curr.NextPtr                    // Continue with lateral search
16 return Node_curr                                     // Contains key K if present
```

performs a downward traversal using the pointer corresponding to `MaxKey` of the previous node. The pointer of the previous node is maintained during lateral traversal for efficient backtracking. The search continues till the data layer L_0. All the keys in a super-node in the data layer are probed using warp-level primitives, and the traversal continues to the next lateral node if K is greater than `MaxKey`; otherwise, K must lie in the current super-node if it is present in the skip list.

Supporting Operations. An insert operation first searches the candidate super-node where K is to be inserted. If the super-node is not full, the node is *locked* and K is added to the super-node; otherwise a new super-node is created, the keys in the candidate super-node are split, and half of the keys from this node are transferred to the new node. The key K is added to either the old candidate node or the newly created node based on the sorted order of the keys. After the split, the keys in the old candidate super-node are raised to the index layer L_1. Algorithm 3 shows the pseudocode for inserting a (key,value) pair to a skip list. For search, once a candidate node is identified, warp-level intrinsics are used to compare the data fields of the super-node with the input key K in parallel. The corresponding value is returned if K is found; otherwise the search returns a negative value denoting absence of K in the skip list.

A delete operation on key K needs to handle three scenarios: (a) remove K from the candidate super-node (say $Node_c$) without a need for merging super-nodes, (b) merging super-nodes is required on deletion, and (c) K is found in the last node at a level. In case (a), K is deleted, and the subsequent keys in $Node_c$ are shifted to their left atomically to fill the vacant slot. The `MaxKey` field is updated prior to deletion of K for correctness. In case (b), a merge operation is performed when the removal of K reduces the number of keys in $Node_c$ to *one-third* of the maximum super-node capacity. All the remaining keys in $Node_c$ are merged with the next super-node in the same layer (say $Node_{c+1}$), and $Node_c$ is marked for physical deletion. If $Node_{c+1}$ cannot accommodate the keys from $Node_c$, the super-node $Node_{c+1}$ is split. Half of the keys of $Node_{c+1}$ are moved to the newly created node arising from the split, and keys from the candidate node $Node_c$ are copied

Algorithm 3: Insert Operation in SL-UVM

Input: Skip list SL, key K, and value V
Output: A Boolean variable indicating the status of the operation

```
1  Node_c ← FindCandidateNode(SL, K)                    // Search for the candidate super-node
2  if K ∈ Node_c then
3      return false                                      // Update the corresponding value V
4  if ¬isSuperNodeFull(Node_c) then
5      Lock(Node_c)
6      InsertSorted(Node_c, K, V)                        // The keys in Node_c are kept sorted
7      Unlock(Node_c)
8      return true
9  else                                                  // Super-node is full and a split is required
10     Node_new ← AllocateNewSuperNode()
11     Lock(Node_c); Lock(Node_new)
12     midKey ← GetMedianKey(Node_c)                     // Transfer half of keys to the new node
13     MoveUpperHalfKeys(Node_c, Node_new)               // Move from Node_c to Node_new
14     if K < midKey then                                // Insert into the appropriate node in sorted order
15         InsertSorted(Node_c, K, V)
16     else
17         InsertSorted(Node_new, K, V)
18     Node_new.NextPtr ← Node_c.NextPtr; Node_c.NextPtr ← Node_new
19     PromoteKeys(Node_c, L_1)                          // Old keys are raised to index layer L_1
20     Unlock(Node_new); Unlock(Node_c)
21     return true
```

to the newly-created super-node. In case (c), if K is the last remaining key in the last node of the level, the key K is deleted without marking the candidate node for physical deletion.

UVM Support. To enable a vanilla UVM support, we incorporate managed memory allocation for the super-nodes of GFSL. We also experiment with the memory advise and prefetch hints, and find that only prefetch hints are useful for the super-nodes of SL-UVM and the input key array. We do not use any hints for the auxiliary value array.

```
// Hints for the data structure DS (either hash table or skip list)
cudaMemPrefetchAsync(DS, ...);
cudaMemAdvise(DS, ...); // Explore all three possible advises
// Hints for the input array of keys and values kv_pairs
cudaMemPrefetchAsync(kv_pairs, ...);
cudaMemAdvise(kv_pairs, ...);
launchKernel<<<...>>>(DS, kv_pairs, ...); // insert, delete, and search
```

Listing 1. Sample driver to show application of memory advise and prefetch hints

Listing 1 shows a generic driver to execute the insert, delete, and search operations on the HT-UVM and SL-UVM designs. The code snippet shows the locations of the CUDA memory advise and prefetch hints in the driver.

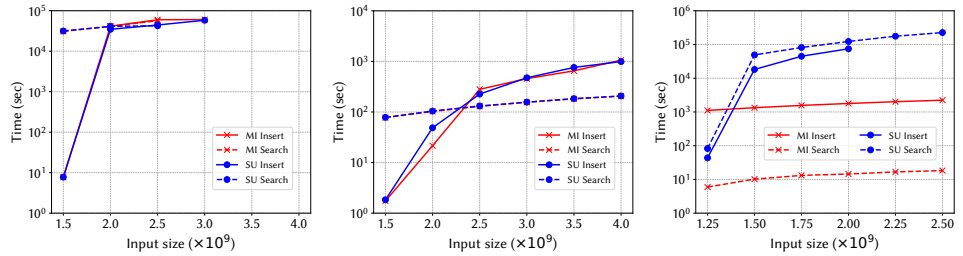
3.3 Shortcomings of the Vanilla UVM Design

To study the baseline UVM designs, we evaluate how the `insert()` and `search()` kernels in HT-UVM and SL-UVM scale with varying degrees of oversubscription. Figure 4 shows the performance results for two input sequences. The sparse unique (SU) input has unique keys randomly distributed over the allowed range of 32-bit keys. The monotonically increasing (MI) input has unique keys in strictly increasing order. All the run times are shown in log scale. The x-axis shows the length of the input key sequence that is operated on, with 2×10^9 and 1.25×10^9 keys roughly saturating the GPU global memory for the HT-UVM and SL-UVM designs respectively. Beyond 2×10^9 keys, the GPU memory is oversubscribed for HT-UVM and the 100% oversubscription point is reached with 4×10^9 keys (ignoring the auxiliary value array). Thus, 2.5×10^9, 3×10^9, and 3.5×10^9 keys correspond to respectively 25%, 50%, and 75% memory oversubscription levels for HT-UVM. For the same input size, SL-UVM has a bigger memory footprint than HT-UVM arising from maintaining the down pointers between the skip list levels and the partially filled super-nodes created due to split operations invoked in the `insert()` kernel (lines 10–13 in Algorithm 3). For SL-UVM, the GPU memory is oversubscribed beyond 1.25×10^9 keys and the 100% oversubscription point is reached with 2.5×10^9 keys. Thus, 1.5×10^9, 1.75×10^9, 2×10^9, and 2.25×10^9 keys correspond to respectively 20%, 40%, 60%, and 80% memory oversubscription levels for SL-UVM.

Figure 4a shows the run time of the `insert()` and the `search()` kernels for HT-UVM without data placement and prefetch hints. When there is no oversubscription, i.e., input size is 1.5×10^9, the `insert()` kernel completes within seconds with both inputs. However, the `search()` kernel suffers from far faults even for an input of 1.5×10^9 keys because the search operation is done on a pre-built hash table having 4×10^9 keys, which overflows the GPU memory. We employ a pre-built hash table for search workloads to effectively capture diverse access patterns across the traces evaluated in our study. Additionally, any locality in the input stream of the search operation gets destroyed by the hash function which can map even consecutive keys to different pages of the hash table. In contrast, the `insert()` kernel gradually grows the hash table and hence, for 1.5×10^9 keys, the hash table pages remain resident in GPU memory. But once oversubscription sets in (i.e., 2×10^9 keys onward), the time taken by both kernels on both inputs runs into hours. Both kernels fail to complete for 3.5×10^9 and 4×10^9 input keys even after 20 hours, which we use as the timeout bound for all our hash table experiments.

Figure 4b shows that the run time of both the kernels for HT-UVM with data placement and prefetch hints reduces from hours to minutes, underscoring the importance of these hints in UVM-enabled designs. Even the inputs having 3.5×10^9 and 4×10^9 keys complete under half an hour. However, the run time of the `insert()` kernel continues to grow exponentially in input size. Its run time increases from under hundred seconds to thousand seconds as the input size increases from 2×10^9 keys to 4×10^9 keys. The run time of the `search()` kernel scales relatively slowly compared to the `insert()` kernel, but the growth is still superlinear.

Figure 4c shows a similar trend for SL-UVM in the presence of the data placement and prefetching hints. With the SU input, the `search()` kernel is more expensive than the `insert()` kernel up to 2×10^9 keys, beyond which point the `insert()` kernel ends up spending most of its time waiting for far faults to resolve and fails to complete within 72 hours, which we use as the timeout bound in all our skip list experiments. Importantly, the difficulty of scaling the footprint of a skip list is also reflected in the scale of the y-axis of Figure 4c, which runs into several tens of hours. The difference between the `insert()` and the `search()` kernels for SL-UVM is that the latter works on a pre-built skip list. With oversubscription, the `search()` kernel suffers from poor locality from the very beginning of execution because the search key could lie anywhere in the pre-built skip list. On the other hand, the `insert()` kernel enjoys comparatively better locality and does not encounter



(a) HT-UVM without hints (b) HT-UVM with hints (c) SL-UVM with hints

Fig. 4. Scaling of HT-UVM and SL-UVM with oversubscription

far faults until the skip list starts overflowing GPU memory. For the MI input, the `insert()` and `search()` kernels' run time on SL-UVM scales relatively slowly compared to the SU input. The `insert()` kernel takes minutes and the `search()` kernel takes seconds to complete for the MI input. The locality in the input key stream present in the MI input leads to significant reduction in the run time for the kernels. The working set of a contiguous batch of keys in the input is limited to a part of the skip list, thereby providing good performance. The `search()` kernel outperforms the `insert()` kernel for the MI input by utilizing the parallel resources of the GPU due to absence of locks. The `insert()` kernel's performance remains limited by lock contention despite good locality in the MI input. We present techniques to alleviate this lock contention in Section 4.2. Overall, both the `insert()` and `search()` kernels show superlinear growth in run time as the input size increases pointing to significant scalability bottlenecks.

HT-UVM and SL-UVM suffer from poor scalability whenever there is a lack of locality in the input key pattern leading to accesses going to distant parts of the data structure within a short time window. With oversubscription, this increases the possibility of the requested page being not resident in GPU memory, leading to frequent far faults. Additionally, the hash function in HT-UVM may map even consecutive keys to buckets resident in different pages, thereby destroying any locality present in the input key stream.

To further study the benefits of the data placement and prefetching hints, Figure 5 quantifies the volume of far faults observed by the `insert()` kernel of HT-UVM running on the MI input without any hint. For comparison, it also shows (along the right y-axis) the volume of remote mapping events (a reasonable proxy for far faults) when hints are enabled. The number of page faults is significantly reduced with hints. The hint `cudaMemAdviseSetAccessedBy`, used for the hash table pages, does not trigger data migration, but only sets up the virtual to physical address translation entries in the GPU page table [54]. Instead of migrating pages, this hint allows the GPU to access the hash table pages from the CPU memory through the communication network (e.g., PCIe or NVLink), thereby lowering the degree of GPU memory thrashing. These accesses lead to remote mapping events and do not manifest as page faults. This is why the volume of remote mapping events captures the overhead of accessing the hash table pages from the system memory. However, remote mapping helps only when the degree of GPU memory oversubscription is small because remote accesses are faster than serving frequent page faults by the driver [28]. As the input sequence length increases, the increasing degree of memory oversubscription results in a linear increase in the volume of remote mapping events, since a larger fraction of GPU pages are evicted. This, in turn, increases the communication and contention on the PCIe/NVLink interconnect leading to overall poor scalability of the HT-UVM design *even* in the presence of the data placement and prefetching hints. As a result, frequent use of remote mapping events may result in poorer

performance compared to serving page faults in certain situations. Thus, the optimal set of hints for a data structure *may* depend on the input pattern as well as the input sequence length, adding an *extra* layer of complexity to program analysis techniques [10, 28].

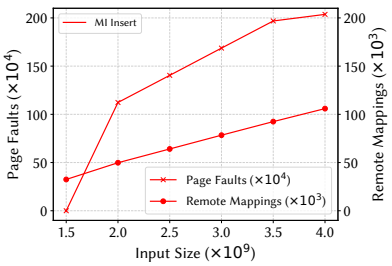
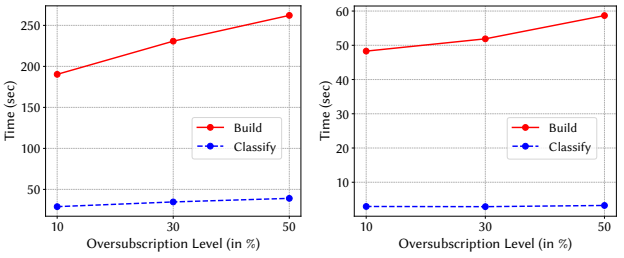


Fig. 5. Scaling of page faults and remote mapping events in HT-UVM



(a) Scaling with HT-UVM (b) Scaling with SL-UVM

Fig. 6. Scaling of a metagenomics classifier application [36] that uses HT-UVM or SL-UVM with memory oversubscription

We further study the scalability of a real-world application that uses HT-UVM and SL-UVM to implement key-value stores. Figure 6 shows the scaling of a metagenomics classifier [36] with data placement and prefetching hints enabled. Both HT-UVM and SL-UVM exhibit *poor* scaling behavior in the two key phases of the metagenomics classifier, namely, Build (uses the `insert()` kernel) and Classify (uses the `search()` kernel). As the degree of memory oversubscription increases from 0% to higher levels, the run time of both the phases increases non-linearly. Without data placement and prefetching hints, the application run time degrades dramatically, reaching the order of hours.

These results demonstrate that far page faults and page migration significantly impact the performance of real-world applications under memory oversubscription, highlighting their importance in everyday use cases. Existing techniques, such as prefetching, page placement hints, and page replacement policies, optimize memory usage, but often fail to be effective when applications with irregular access patterns oversubscribe GPU memory [3, 5, 21, 75, 76]. Applications with irregular access patterns, e.g., graph-based applications and concurrent GPU data structures that require pointer chasing, have poor locality, making prefetching counterproductive. Thus, it is imperative to come up with targeted novel data structure designs that can scale with the input size.

4 Designing Oversubscribed Data Structures

Scaling HT-UVM and SL-UVM to high degrees of GPU memory oversubscription requires novel designs that take into consideration the potential lack of locality in the input access patterns. In the following, we discuss the design of a hash table and a skip list that scales better with input size.

4.1 Hash Tables with Memory Oversubscription

The key insights for minimizing page faults are to (i) group nearby keys in the input sequence to enhance the locality of the input, and (ii) translate the locality, if any, present in the input sequence to the hash table architecture. The first requirement ensures that keys within the same page are processed together (i.e., near in time) leveraging spatial locality. The second requirement implies that the nearby keys map to nearby memory pages of the hash table. This is, in general, challenging because a general hash function may end up mapping even consecutive keys to far apart pages. Taken together, the aforementioned two requirements optimize memory access pattern and facilitate efficient prefetching, which, in turn, reduces the frequency of on-demand page faults and subsequent stalls during operations improving overall performance.

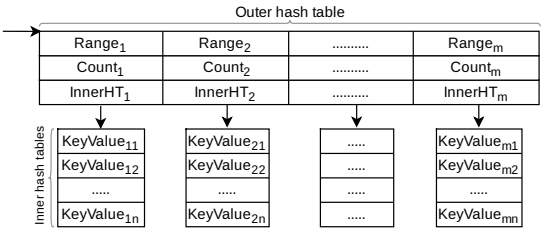


Fig. 7. Layout of HT-OVS showing its hierarchical structure

Keeping the aforementioned two requirements in mind, we propose a two-level hash table structure, denoted by HT-OVS, where the outer hash table categorizes keys into *ranges* while each inner hash table stores the actual (key, value) pairs belonging to the corresponding key range. Figure 7 shows the layout of our proposed design. We design the outer hash table such that the keys mapping to an outer hash table entry belongs to a contiguous range, satisfying the first requirement. The entire range of keys captured by one outer hash table entry gets co-located within the pages of one inner hash table, thereby satisfying the second requirement as well.

The outer hash table is allocated using `cudaMallocManaged()`, the UVM managed memory allocation API of CUDA. The number of entries in it is determined as `MAX_KEY/INNER_HT_SIZE` at run-time, where `MAX_KEY` represents the maximum value a key can take and `INNER_HT_SIZE` is the number of keys one inner hash table can accommodate. Each outer hash table element contains three fields: `InnerHT`, `Range`, and `Count`. The field `InnerHT` contains a pointer to an inner hash table of size `INNER_HT_SIZE` that stores the actual (key, value) pairs. The `Range` field stores the numeric range of the keys stored in the inner hash table pointed to by the `InnerHT` field. The hash function used to map a key K to an outer hash table entry is $h_o(K) = K/\texttt{INNER\_HT\_SIZE}$. The `Count` field is the count of the unique keys present in the inner hash table pointed to by the `InnerHT` field. This `Count` helps maintain the load factor of the inner hash table. The inner hash table uses the same hash function and collision resolution schemes as the baseline HT-UVM design.

Given an input sequence of operations, one can launch the corresponding kernel with the entire sequence of operations as input or launch the kernel iteratively with each invocation accepting a smaller batch of inputs. HT-OVS uses the latter option by default because smaller batches lead to a smaller number of concurrent operations and reduced memory thrashing. Thus, smaller batches allow retaining more pages of the large hash table structure in the GPU global memory, thereby reducing the number of far faults. Sending the whole input sequence at once would likely evict more hash table pages from the GPU memory and trigger far faults, thereby reducing performance.

The HT-OVS design described thus far works well for input sequences with small strides between keys, resulting in fewer ranges being touched and all ranges being reasonably populated. But for sequences where the keys have large strides (i.e., spans across multiple ranges), accessing distant ranges sequentially leads to touching multiple inner hash tables with little or no locality, increasing the volume of page faults. Preprocessing the input sequence by sorting the keys allows us to increase locality [44, 77]. However, sorting the entire input sequence requires knowing the whole sequence in advance, which may not always be feasible. Sorting a small batch of input keys at a time only requires the knowledge of a single batch before insertion, which is particularly useful in streaming scenarios where the input arrives in real-time. Sorting each batch of the input by key also satisfies the second requirement of enhancing locality by grouping nearby keys.

HT-OVS supports the basic operations: `insert()`, `delete()`, and `search()`. A co-operative group of threads is responsible for operating on a (key, value) pair (x, y). The hash of the x is used for looking up the outer hash table. A hit indicates the presence of at least one key in the corresponding inner hash table belonging to the same range as x. Next, the `InnerHT` is used to access the inner hash table and locate the key x. The `Count` field in the outer hash table slot is incremented by one if the key is inserted for the first time as indicated by the outcome of the `insert()` operation on the inner hash

Algorithm 4: Insert Operation in Two-level HT-OVS

Input: Hash table *OuterHT*, Key K, and Value V

1 $index \leftarrow K/INNER\_HT\_SIZE + 1$ `// Locate the index in outer hash table`
2 **if** $OuterHT[index].Range = NULL$ **then** `// Inner hash table is empty`
3 $OuterHT[index].Range \leftarrow index$ `// Update range tracked by outer hash table`
4 Call Algorithm 1 `// Insert` (K, V) `in inner hash table`

table. An insert operation in HT-OVS follows two steps, as shown in Algorithm 4. For a key-value pair (K, V), the algorithm first identifies the range the key belongs to (denoted by *index*). In other words, this helps in identifying the appropriate inner hash table. The entry at $OuterHT[index]$ in the outer hash table tracks all the keys K in the range $(index - 1) \times$ `Inner_HT_Size` $\leq K \leq index \times$ `Inner_HT_Size` $- 1$. A value of zero for `Range` implies an empty range. The `Range` and `Count` fields of $OuterHT[index]$ are updated. We update the value of `Range` using warp intrinsics if the key being inserted is the first key in that range. After finding the appropriate inner hash table, we call the same core insert function (Algorithm 1) on the inner hash table. The only difference is that the corresponding field `Count` in the outer hash table is atomically incremented after a successful insert operation in Algorithm 1.

The search operation also follows a two-step process. First, the warp associated with the query key identifies and probes the appropriate outer hash table. Once the corresponding inner hash table is determined, the warp invokes the core search function on the inner hash table. The two-level search strategy provides a distinct advantage over a conventional search operation, such as that employed in HT-UVM, particularly for negative queries (i.e., when the searched key is absent from the hash table). If the outer hash table indicates that no key exists within the range which the queried key maps to, the system can immediately terminate the search and return a negative result, thereby avoiding the invocation of the core search kernel. This early-exit optimization significantly reduces unnecessary computation and improves overall search efficiency in certain cases.

4.2 Skip Lists with Memory Oversubscription

Unlike a hash table, mutation operations on a skip list using a lock-based algorithm like GFSL has substantially less parallelism. For example, Figure 4c shows that performing 2×10^9 insertions in a skip list for the SU input takes several hours. To understand the performance bottlenecks of the `insert()` kernel of SL-UVM, we profile it with the NVIDIA Nsight profiler [56] and collect application level statistics of the kernel for three different inputs (SU, DU, and MI)[4], as shown in Table 1 (for 2.5×10^9 keys, the `insert()` kernel on the SU input runs out-of-memory due to the memory overhead from tracking statistics). In the following, we first discuss the bottlenecks identified through this exercise and the optimizations to alleviate them in the basic GFSL algorithm. Next, we present our two-level skip list design optimized for memory oversubscription. We refer to our optimized skip list design as SL-OVS.

4.2.1 Improvements to GFSL. In this section, we discuss the characteristics/bottlenecks identified in the GFSL design and the optimizations to address these bottlenecks or exploit these characteristics.

Pinning Index Layers. A key K residing in level L_i is raised to level L_{i+1} only when the supernode containing K is split during an insert operation. As a result, the expected height of GFSL

[4]The DU input is constructed by first dividing the entire key range into smaller blocks and then randomly sampling unique keys from each block such that the keys drawn from a particular block appear contiguously in the input sequence, thereby ensuring some amount of intra-block locality.

Table 1. Profiled statistics of the `insert()` kernel of SL-UVM for three different inputs

Input	#Keys ($\times 10^9$)	Lock acquire attempts	L_0 nodes	Lateral movement	#Keys ($\times 10^9$)	Lock acquire attempts	L_0 nodes	Lateral movement
SU		1.00	95.24	1.60		1.00	95.24	1.64
DU	1.25	1.00	95.24	1.63	1.5	1.00	95.24	1.60
MI		10.63	94.24	1.27		10.58	94.24	1.25
SU		1.00	95.24	1.68		1.00	95.24	1.57
DU	1.75	1.00	95.24	1.65	2	1.00	95.24	1.56
MI		11.52	94.90	1.53		10.58	94.24	1.22
SU		1.00	95.24	1.66		x	x	x
DU	2.25	1.00	95.24	1.57	2.5	1.00	95.24	1.55
MI		10.58	94.24	1.21		10.60	94.24	1.21

and SL-UVM is less compared to that in the traditional skip list. Table 1 shows the percent of super-nodes in data layer (column "L_0 nodes") of the skip list. Across all inputs, close to 95% of all the super-nodes allocated are in the data layer. Given the small fraction (about 5%) of super-nodes in the index layers, pinning them in the GPU global memory will ensure that the pages holding the top layer super-nodes will never get evicted, thereby avoiding faults to these pages. This is important because the super-nodes in the upper layers of the skip list are accessed more frequently on average compared to the data layer. To incorporate this optimization in SL-UVM, we use different memory pools for allocating the super-nodes for the data layer and the index layers. We use `cudaMallocManaged` for the data layer so that the skip list can oversubscribe GPU memory, while we allocate super-nodes in the index layer using `cudaMalloc`.

Separate Memory Pool for Each Layer. The column "Lateral movement" of Table 1 lists the ratio of the number of lateral pointer traversals (intra-layer) to the number of downward pointer traversals (inter-layer) during the insert operations. For the SU and DU inputs, the lateral traversals are approximately 1.6× more prevalent than downward traversals, while for the MI input, this is slightly less but at least 1.2× across different input sizes. A larger fraction of lateral traversals highlights the fact that super-node accesses remain confined to the same layer more often. Therefore, to reduce page faults during lateral traversals within an index layer, it is more judicious to allocate the super-nodes of that layer from a single memory pool as opposed to allocating the super-nodes of all index layers from a single memory pool haphazardly. We explore the idea of a dedicated contiguous memory pool for each index layer to exploit the access locality arising from the predominantly large volume of lateral pointer traversals. As already discussed, the memory pools for the index layers do not use managed memory and are allocated using `cudaMalloc` to keep the page pinned to GPU memory.

Unsorted Keys in the Data Layer. GFSL maintains keys in the sorted order within each super-node at each layer. Sorting of keys incurs substantial overhead due to frequent shifting of the keys within a super-node via expensive global atomics during each insert operation. Maintaining keys in unsorted order in a super-node improves the `insert()` kernel performance. Sorting of keys in a super-node is, however, required during the split of a full super-node. The cost of sorting the keys on infrequent splits can potentially be overshadowed by the savings from not sorting the keys on every insertion operation.

Reducing Lock Contention. Profiling the performance of GFSL on a sorted input sequence (e.g., MI) exposed another bottleneck. The algorithm does not scale well for a sorted input sequence due to lock contention. Since the input keys are sorted, all the warps proceed more or less together while building a skip list through insertion operations. As a result, almost all warps contend to acquire the lock of each super-node as the skip list grows. The column headed "Lock acquire attempts" in Table 1 shows the total number of lock acquire attempts per successful lock acquire. While this is close to unity for the SU and DU inputs, at least ten locking attempts are needed on average per lock acquire for the MI input highlighting significant lock contention. Our profile study further observes that this lock contention primarily arises from assigning consecutive keys from the sorted MI input sequence to consecutive warps leading to all warps contending for the lock of the next super-node to be added to a growing skip list. This key-to-warp assignment algorithm will be referred to as the cyclic distribution algorithm. To reduce the lock contention, we explore the impact of a block-cyclic distribution of keys to warps. In this algorithm, blocks of consecutive keys are assigned to warps in a cyclic manner, i.e., block n is assigned to warp ($n \mod w$), where w is the total number of warps. We present the study for deciding the optimal block size (in terms of number of keys) and the optimal grid configuration in Section 5.3.

4.2.2 Improving Performance under Oversubscription. Similar to hash table, we propose a two-level hybrid skip list data structure to efficiently handle memory oversubscription by introducing locality in the operations. Figure 8 shows the design of the proposed two-level hierarchical structure (referred to as SL-OVS). The outer hash table tracks the ranges of keys present in an inner skip list. Each inner skip list is an SL-UVM. This two-level structure further improves concurrency and reduces lock contention for the skip list.

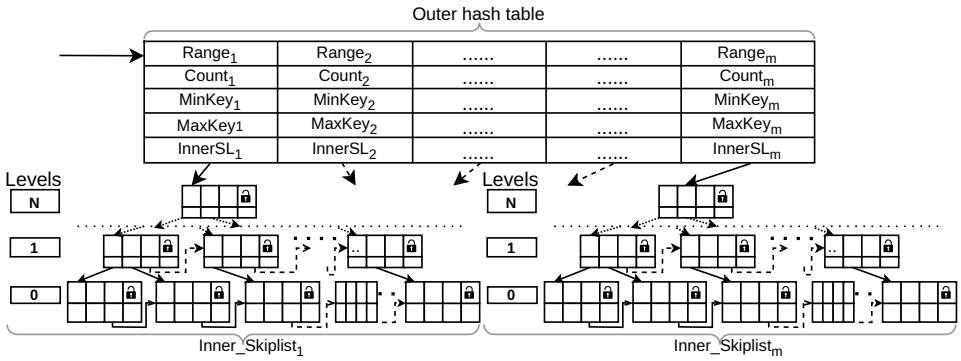


Fig. 8. Layout of SL-OVS

Operations in SL-OVS. Similar to GFSL, each operation in SL-OVS is assigned to a warp. The entire key range is divided into equal-sized groups and each slot in the outer hash table tracks a group. Each outer hash table entry maintains a pointer (`InnerSL`) to the corresponding inner skip list, the range of keys in the inner skip list (`Range`), the number of unique keys in the inner skip list (`Count`), and the minimum and the maximum keys (`MinKey` and `MaxKey`) in the inner skip list. A procedure similar to Algorithm 4 is followed for SL-OVS operations. The actions on the outer hash table remains the same, except the additional updates of `MinKey` and `MaxKey` of a range on insert and delete operations. Different keys from different ranges can now be processed concurrently by different warps by jumping to different inner skip lists after looking up the outer hash table, thereby improving concurrency and reducing lock contention.

Table 2. Description of inputs used for evaluation

MI	Monotonically increasing sequence of keys in range $[1, \texttt{NUM\_INS}]$
DU	Random keys in range $[1, \texttt{NUM\_INS}]$ without repetition, randomization is within blocks of $[i, (i+1) \times 5 \times 10^8]$, $i \geq 0$ (Dense Unique)
DR	Random keys in range $[1, \texttt{NUM\_INS}]$ where 20% of the keys are repeated, randomization is within blocks of $[i, (i+1) \times 5 \times 10^8]$, $i \geq 0$ (Dense Repeat)
SU	Random keys in range $[1, \texttt{UINT32\_MAX}]$ without repetition (Sparse Unique)
SR	Random keys in range $[1, \texttt{UINT32\_MAX}]$ with 20% repetition (Sparse Repeat)
SUR	Random keys in range $[1, \texttt{UINT32\_MAX}]$ without repetition, randomization is within blocks of $[i \times 5 \times 10^8, (i+1) \times 5 \times 10^8]$, $i \geq 0$ and the blocks are shuffled (Sparse Unique Random)
DUL	Random keys in range $[1,$ nearest 10^9 to $\texttt{NUM\_INS}]$ without repetition (Dense Unique-like)
ZZ	Numbers present in zigzag pattern of $1, \texttt{UINT32\_MAX} - 1, 2, \texttt{UINT32\_MAX} - 2, 3, \cdots$ (Zigzag)

Finding Predecessor in SL-OVS. The expected cost of performing each operation in SL-OVS is logarithmic in the number of keys in the skip list. However, finding the predecessor of $\texttt{MinKey}_r$, i.e., the minimum key of the inner skip list maintaining the keys in range r, requires identifying the previous non-empty range in the outer hash table. To do this efficiently, SL-OVS maintains an auxiliary skip list of non-empty ranges encountered so far. The previous non-empty range $r' = predecessor(r)$ can be located efficiently in the auxiliary skip list, and the maximum key of the range r' ($\texttt{MaxKey}_{r'}$) is returned from the outer hash table slot of r' as the predecessor of $\texttt{MinKey}_r$, i.e., $predecessor(\texttt{MinKey}_r)$ is $\texttt{MaxKey}_{predecessor(r)}$. The *successor* operation is supported similarly.

5 Evaluation

In this section, we discuss our evaluation infrastructure and present experimental results of our implementations with different input sequences. Since this is the first work to explore GPU data structures with memory oversubscription, there is no prior work that we can directly compare with. We extend publicly available in-memory GPU hash tables such as cuCollections [49], Stdgpu [67–69], and GPH [14] to support memory oversubscription and compare the extended implementations with our HT-OVS proposal. We also evaluate our design with implementations optimized with SUV [10], a tool for automatic insertion of prefetching and data/translation placement hints relevant to workloads that oversubscribe GPU memory.

5.1 Evaluation Infrastructure

Input Sequences. We evaluate our design proposals with eight different input sequences each capturing a distinct key access pattern, as listed in Table 2. The MI, DU, and DR input sequences contain keys in the range $[1, \texttt{NUM\_INS}]$, where NUM_INS is the total number of insert operations in the input sequence. The keys appear in a strictly increasing order in the MI sequence, while the order of keys is random in DU and DR. The input key range for the SU and SR sequences is $[1, \texttt{UINT32\_MAX} - 1]$, with keys appearing in a random order. The sequences with repetition (DR and SR) have 20% keys duplicated. The DUL input sequence is DU-like, but the keys are shuffled in batches of size 10^9. In SUR, the keys are randomized within batches of size 5×10^8 and the batches are shuffled among themselves. In ZZ, the keys are arranged in a zigzag manner.

Input Preprocessing. We sort the input sequences in ascending order to improve spatial locality, as in prior work [44, 77]. Sorting the entire input is not feasible for a streaming input. Hence, we employ batching. The input sequence is divided into smaller batches, such that the entire footprint of the constructed data structure and the input batch fits in the GPU global memory.

Table 3. GPU system configuration

GPU	NVIDIA Quadro RTX 5000	**OS**	Ubuntu 22.04.1
GPU Memory	16 GB	**Kernel**	Linux 6.8.0-59-generic
Driver Version	580.76.05	**Compiler**	`nvcc -std=c++17 -arch=sm_75 -O3`
Graphics Bus	PCI Express 3.0 x 16		

Both HT-OVS and SL-OVS sort every batch of keys as a performance optimization before executing the input sequence of operations on the data structure. We use the Thrust library [55] for GPU-based sorting, as it offers better performance compared to the CPU-based sorting methods. The execution time presented for HT-OVS and SL-OVS includes the time spent in sorting, ensuring that the reported performance includes end-to-end execution of the operation sequence for fairness.

GPU Device and Toolchain. Table 3 lists the specifications of the discrete GPU and the compiler toolchain used in our evaluation. In addition to native performance runs, we use two profilers for analyzing the performance. These are Nsys version 2025.1.3.140 and NVProf version 12.9.19. Although NVProf is deprecated, we use it to get an estimate of the number of remote mapping events when employing the `cudaMemAdviseSetAccessedBy` hint, as described in Section 3.3.

Table 4. Software parameters for hash table

`CGS`	Cooperative group size	16
`GBS`	GPU batch size for HT-UVM	10^8
`GBS`	GPU batch size for HT-OVS	10^9
`RNG`	Inner hash table size	2^{20}

Table 5. Software parameters for skip list

`TPB`	Threads per block	512
`GBS`	GPU batch size	2.5×10^8
`RNG`	Inner skip list size	2^{21}
`KPB`	Keys per block in block-cyclic distribution	300

Tunable parameters. Tables 4 and 5 list the default values of the tunable parameters for the hash table and skip list experiments, respectively. For hash table, `CGS` controls the number of threads in a cooperative group. `GBS` controls the numbers of elements processed in each kernel call, while `RNG` controls the size of inner hash table.

Unlike hash tables, the performance of the skip list is sensitive to the grid configuration which we explore in Section 5.3. We choose a batch size of 2.5×10^8 for skip list controlled by `GBS`, which allows us to explore similar degrees of oversubscription across the two data structures because oversubscription in skip lists sets in with smaller input sizes. `RNG` sets the size of each inner skip list. `KPB` controls the number of keys in a block when block-cyclic distribution of keys among warps is used. The number of entries in the outer-level hash table of SL-OVS is calculated as $\lceil MAX\_KEY\_RANGE/RNG \rceil$.

5.2 Evaluation of Hash Table

Performance Comparison. We compare the performance of HT-UVM and HT-OVS on the `insert()`, `delete()`, and `search()` kernels using the input sequences discussed earlier. Given a kernel, we vary the number of keys input to each kernel from 2×10^9 to 4×10^9 in steps of 5×10^8. For each key, the `insert()` kernel also takes a value as input. For 2×10^9 key-value pairs, the `insert()` kernel's footprint just exceeds the GPU's global memory capacity of 16 GB. As the input size increases to 2.5×10^9, 3×10^9, 3.5×10^9, and 4×10^9, the degree of memory oversubscription grows to 25%, 50%, 75%, and 100%, respectively. As we discuss in the following, our design efficiently handles this progressively increasing memory oversubscription.

Figure 9 shows the performance of HT-OVS normalized to HT-UVM for the `insert()` and `delete()` kernels. For each oversubscription level, the bars correspond to the different input sequences listed in Table 2. These results show that the speedup achieved by HT-OVS increases substantially for all the input sequences as the degree of oversubscription increases. For example, with the SR input, the speedup increases from 2.3× to 21.9× as the oversubscription grows from 25% to 100% for the `insert()` kernel. The DUL input enjoys the maximum speedup for most of the input sizes. Unlike the `insert()` kernel, the speedup does not vary with the degree of oversubscription for the `delete()` kernel. With the SR input, the speedup remains more or less constant at around 8×. Except for SR, SU, and DU, the rest of the input sequences enjoy a speedup of around 20× for all input sizes.

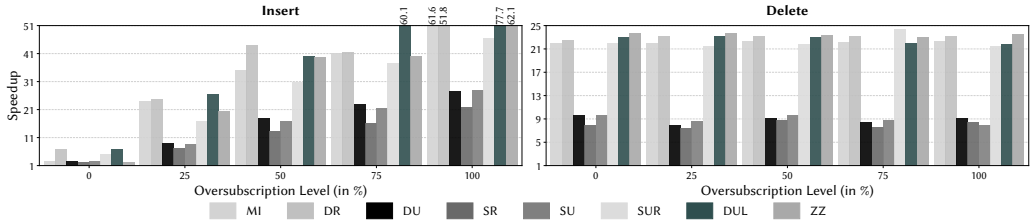


Fig. 9. Speedup of HT-OVS relative to HT-UVM for the `insert()` and `delete()` kernels.

Figure 10 evaluates the most oft-used `search()` kernel. The left figure shows the speedup achieved by HT-OVS when all search queries hit in the hash table (referred to as positive queries). The speedup ranges from about 5× to 17× across different input sizes and input sequences. Figure 10 also shows an evaluation of our design when the input sequences query for keys that are not resident in the hash table (referred to as negative queries). The three groups of bars use negative queries constituting one-third, one-fourth, and one-fifth of the total search queries provided. HT-OVS is able to achieve a speedup of about 2× to 7× even in the presence of negative queries. A negative query requires searching the hash table until an unoccupied entry is encountered or the end of the hash table is reached. The HT-OVS design is able to constrain the search space within an inner hash table making negative queries more efficient than HT-UVM.

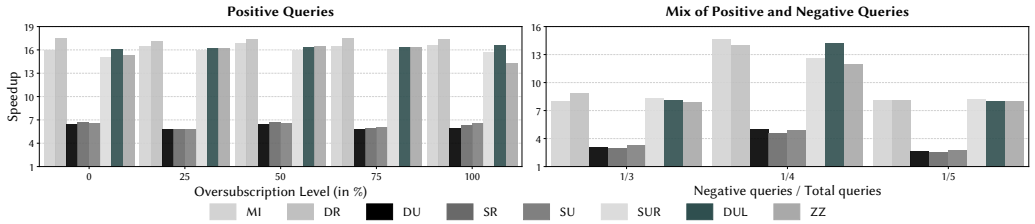


Fig. 10. Speedup of HT-OVS relative to HT-UVM for the `search()` kernel with all positive queries and with a mix of positive and negative queries.

Sensitivity Study. In the following, we evaluate the sensitivity of HT-OVS to the three different design parameters listed in Table 4, `CGS`, `GBS`, and `RNG`. Since the dense inputs show maximum sensitivity to these parameters, we conduct this study on the `insert()` kernel with the `DU` input having 4×10^9 keys.

The left subfigure of Figure 11 shows that HT-OVS outperforms HT-UVM for all cooperative group sizes (CG_n refers to a cooperative group of n threads). The right subfigure quantifies the

execution time of HT-OVS with different cooperative group sizes normalized to CG_{16}. Even though CG_4 narrowly outperforms CG_{16} for the case of zero oversubscription, CG_{16} is the best in all other cases and on average, making it the most suitable default configuration.

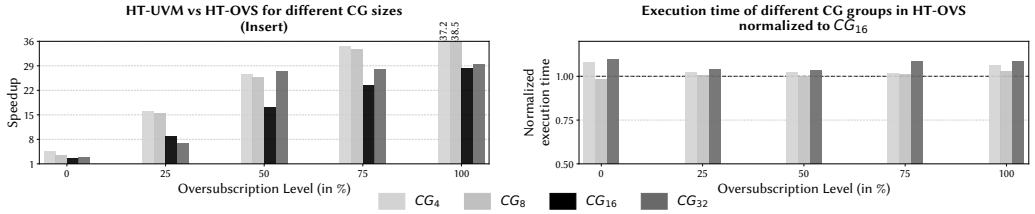


Fig. 11. The left figure shows the speedup of HT-OVS relative to HT-UVM for the `insert()` kernel with varying cooperative group (CG) sizes. The right figure shows the execuion time of HT-OVS with CG_4, CG_8, and CG_{32} normalized to CG_{16}.

Table 6 shows the execution time (in seconds) of the `insert()` kernel as the batch size is varied for both HT-UVM and HT-OVS. The best batch size for HT-UVM is 10^8, while that for HT-OVS is 10^9. Table 7 shows the execution time (in seconds) of the `insert()` kernel as the number of entries in the inner hash table is varied. The execution time decreases monotonically with increasing inner hash table size until the number of entries reaches 2^{20}–2^{22}. Thus, we set the default size of the inner hash table in HT-OVS to 2^{20} entries.

Table 6. Impact of batch size on the run time (in seconds) of `insert()`

Batch Size	HT-UVM	HT-OVS
5×10^7	1311	519
1×10^8	**1007**	364
5×10^8	1280	63
1×10^9	1494	**37**
1.25×10^9	1653	172

Table 7. Impact of the inner hash table size on the run time (in seconds) of the `insert()` kernel for HT-OVS

# Inner Entries	Time	# Inner Entries	Time
2^{12}	54.2	2^{22}	36.5
2^{15}	49.7	2^{24}	38.2
2^{18}	41.1	2^{26}	40.8
$\mathbf{2^{20}}$	36.3		

Sorting Batches in HT-UVM. The HT-OVS design sorts each batch of input keys, while the HT-UVM design does not. We now explore the effect of sorting the input key batches in HT-UVM, similar in spirit to HT-OVS. Figure 12 shows the speedup achieved by HT-UVM *without* sorting over HT-UVM *with* sorting for the `insert()` kernel. These results reveal that HT-UVM does not benefit from sorted batches, rather it suffers a slowdown of at least 1.5× for no oversubscription and approximately 1.3× for small degrees of oversubscription. This is because of two reasons. First, the sorting overhead adds to the end-to-end time. Second, HT-UVM fails to compensate the sorting overhead through improved locality because it cannot translate the locality introduced in a sorted input batch to spatial locality in the hash

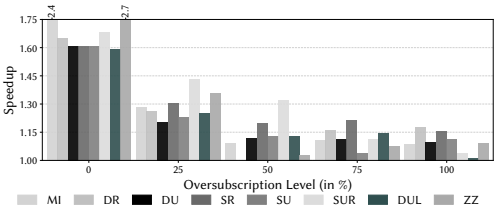


Fig. 12. Speedup of HT-UVM without sorting relative to HT-UVM with sorting for `insert()`

table memory. A hash function can map even consecutive keys to different HT-UVM pages. As a result, sorted batches fail to decrease the number of far faults for HT-UVM. With increasing oversubscription levels, the sorting overhead gets dwarfed by the cost of accessing pages from CPU (remote mapping events as discussed in Section 3.3) leading to a gradually narrowing gap between the performance of the sorted and unsorted implementations. Since HT-UVM performs better without sorting of the input key batches, we use it as our baseline.

Remote Mapping Events. Figure 13 brings to fore the primary reason why HT-OVS is superior to HT-UVM. It shows that the number of remote mapping events experienced by the `insert()` kernel in HT-OVS is at least 3× lower than that in HT-UVM. Thanks to the enhanced locality in the two-level architecture, HT-OVS does not need to use the `cudaMemAdviseSetAccessedBy` hint for the hash table pages. As a result, HT-OVS does not suffer from stalls arising from frequent communication over the NVLink/PCIe. Instead, the two-level architecture allows HT-OVS to utilize GPU memory more effectively so that the hash table pages can be allocated in the GPU memory itself through traditional page faults and still enjoy a significantly lowered total overhead of managing these pages compared to HT-UVM.

Comparison with GPH. We compare the performance of HT-OVS with GPH [14], a recently proposed GPU hash table design employing a perfect hashing technique. However, since GPH is an in-memory hash table and does not support memory oversubscription, we first extend GPH with UVM support and also augment it with suitable data placement and prefetching hints. The `insert()` kernel of the extended GPH implementation can handle input sequences having only up to 5×10^8 keys. Therefore, we simulate 25%, 50%, 75%, and 100% oversubscription levels by reserving portions of the GPU memory so that GPH can access a smaller sized memory, similar to prior work [10]. We also execute HT-OVS with this memory reservation scheme for fairness.

Figure 14 quantifies the speedup achieved by HT-OVS over GPH for three input sequences. HT-OVS achieves a speedup of up to 40.8× over the extended GPH implementation. HT-OVS significantly outperforms GPH for the dense inputs, MI and DU. This improvement stems primarily from the use of batch sorting, which, in the case of dense inputs, renders the key patterns nearly monotonically increasing and generates fewer page faults with HT-OVS. HT-OVS is able to convert the key locality in the inputs into spatial locality of GPU pages with the help of the two-level architecture. However, GPH does not improve with batch sorting due to the same reasons we have already discussed for HT-UVM. The performance improvement of HT-OVS for 100% oversubscription is relatively lower across all inputs due to substantially increased sorting overhead.

We observe that a certain percentage of insert operations fail with GPH in our experiments. These failures arise from the two-phase insert protocol implemented by GPH. In the first phase, GPH tries to insert the keys as in a normal hash table. A fraction of these insert operations may fail in the first phase due to collision. The failed operations from the first phase move on to a refinement phase, where GPH changes the bucket sizes and hash functions to make way for these keys and also rehash the previous keys involved in collision. But this two-phase protocol cannot guarantee failure-freedom. Therefore, a small fraction of keys may not get inserted at all in GPH.

Comparison with cuCollections. CuCollections [49] is an open-source header-only library of concurrent data structures for NVIDIA GPUs. The cuCollections library provides an in-memory hash table implementation called `static_map`. We compare HT-OVS with `static_map` by *minimally* extending the `static_map` implementation[5] to support memory oversubscription. Other than changing the related APIs, we have also introduced a load factor of 0.9 in cuCollections to make

[5] https://github.com/NVIDIA/cuCollections/blob/dev/include/cuco/static_map.cuh

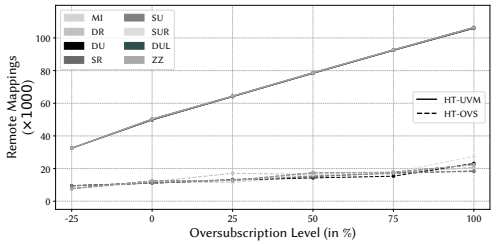


Fig. 13. Number of remote mapping events in the `insert()` kernel for HT-UVM and HT-OVS

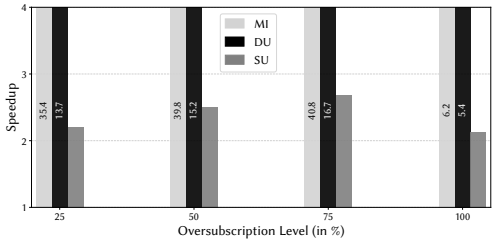


Fig. 14. Speedup achieved by HT-OVS over GPH for the `insert()` kernel

the comparison with HT-OVS fair. We also optimize the extended implementation of `static_map` with data/translation placement and prefetching hints, as discussed in Section 3.1.

Figure 15 quantifies the speedup achieved by HT-OVS relative to the extended `static_map` implementation. For the `insert()` kernel, the speedup increases with increasing levels of oversubscription implying that cuCollections suffers from more faults with increasing input size. The maximum speedup achieved by HT-OVS exceeds 50× for the DUL input.

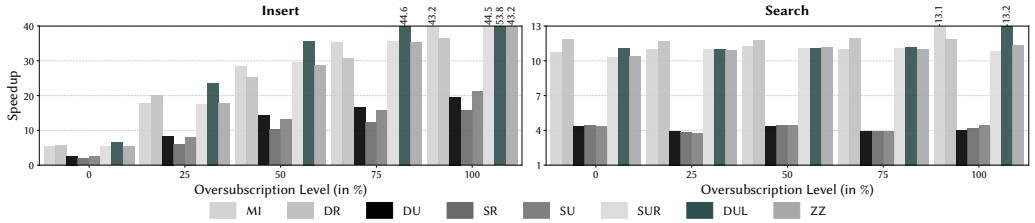


Fig. 15. Speedup achieved by HT-OVS over cuCollections for the `insert()` and `search()` kernels

For the `search()` kernel, HT-OVS achieves 4× to over 13× speedup compared to cuCollections across different degrees of oversubscription. For the `delete()` kernel (not shown in Figure 15), the performance trend is similar to the `search()` kernel, as the algorithm is almost same. HT-OVS achieves maximum speedup in the range of 7× to 26× for the `delete()` kernel with the DR input. The search operation in cuCollections uses asynchronous function calls to return the searched value, which makes it faster than the delete operation.

Comparison with Stdgpu. The Stdgpu [67–69] library implements an open-addressing-based in-memory `unordered_map`.[6] This in-memory hash table of Stdgpu supports insertion of up to 5×10^8 key-value pairs for the GPU used in our evaluation. We extend Stdgpu to support memory oversubscription and compare its performance with HT-OVS for the `insert()` kernel in two oversubscription scenarios, namely, 50% and 75% for three different inputs (MI, DU, and SU). HT-OVS achieves a geomean speedup of 721× (maximum speedup of 1242×) over Stdgpu for 50% memory oversubscription across the three inputs. With 75% memory oversubscription, the geomean speedup of HT-OVS increases further to 868× with a maximum speedup of 1630×. The average execution time of Stdgpu is approximately 3.3× more for the SU input compared to MI and DU inputs. As a

[6] https://github.com/stotko/stdgpu/blob/master/src/stdgpu/unordered_map.cuh

result, HT-OVS enjoys the maximum speedup over Stdgpu for the SU input. The Stdgpu implementation uses several auxiliary structures that contribute to its poor performance by increasing the overall working set size.

5.3 Evaluation of Skip List

We first evaluate the optimizations on baseline GFSL discussed in Section 4.2.1, and then quantify the performance of SL-UVM and SL-OVS. We evaluate our skip list designs using the MI, DU, and SU inputs each with six different sizes ranging from 1.25×10^9 to 2.5×10^9 keys correspoding to 0%, 20%, 40%, 60%, 80%, and 100% memory oversubscription levels.

5.3.1 Evaluation of Baseline Optimizations. We find that the optimizations of pinning the index layer to GPU memory and maintaining separate memory pools for each index layer do not have much positive influence on performance. In fact, maintaining separate memory pools for each index layer leads to a slowdown of about 10% for the `insert()` kernel and 5% for the `search()` kernel. Pinning the index layer to the GPU memory offers a negligible improvement in the `search()` kernel performance up to 60% memory oversubscription, but the `insert()` kernel suffers from a 2% slowdown. The slowdown in these two optimizations arises from the fact that downward pointer traversals are of reasonable volume within the index layers leading to a reasonable volume of page faults on inter-level crossings. We find that the ratio of lateral to downward pointer traversals within the index layers is 1.3, while that in the entire skip list is about 1.6 indicating that a vast majority of the lateral pointer traversals takes place in the data layer. Thus, optimizing allocation of the index layer to favor lateral traversals is not expected to fetch much benefit.

The idea of keeping keys within a super-node unsorted helps the `insert()` kernel, which enjoys an average speedup of 30% across different input sizes. However, this optimization adversely impacts the `search()` kernel performance slowing it down by about 13% on average. Therefore, this optimization is suitable for scenarios involving a much larger proportion of insert operations than search operations.

Next, we study the performance impact of the block-cyclic and cyclic distribution of keys to warps. Table 8 lists the execution time of the `insert()` kernel employing the block-cyclic key to warp assignment with a varying number of keys per block in SL-UVM. These data are collected for the MI and DU inputs with 60% memory oversubscription. In this exploration, the block size is increased in steps of 30 keys to match the capacity of a super-node (30 keys per super-node). Assigning 300 contiguous keys to a warp at a time corresponding to the block size of 300 keys offers the best performance for both the input sequences.

Table 8. Run time (in seconds) of the `insert()` kernel in SL-UVM with varying number of keys per block for block-cyclic distribution of keys

Keys per block		1	30	60	120	150	180	210	240	270	**300**	330	360
Run time	MI	1798	177	109	71	64	59	58	53	54	**49**	49	48
	DU	53	44	46	46	44	44	47	45	45	**46**	47	48

Table 9 shows the sensitivity of the `insert()` kernel performance to the cyclic distribution of keys (i.e., block size is one key) in SL-UVM. In this distribution, each key is assigned to one warp based on the WCWS strategy and this assignment cycles through the warps. The number of warps per thread block is kept fixed at 16 leading to 512 threads per thread block (a warp has 32 threads). This configuration guarantees adequate resource allocation per thread and reasonably good resource utilization [47]. While having many active warps may lead to high contention and

increase in run time due to use of locks, spawning fewer warps may lead to slower progress due to under-utilization of resources. The data show that cyclic distribution of keys with 2^{14} thread blocks outperforms the best configuration of block-cyclic distribution with 300 keys per block in SL-UVM. Therefore, in subsequent evaluations, we use SL-UVM with 2^{14} thread blocks and cyclic distribution of keys. In a similar fashion (details omitted for brevity), we find that the best setting for SL-OVS is block-cyclic distribution with 2^{14} thread blocks and 180 keys per block.

Table 9. Run time (in seconds) of the `insert()` kernel in SL-UVM with varying number of active thread blocks for cyclic distribution of keys

# Thread Blocks	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
Run time for MI	61.06	50.93	50.26	49.28	**45.94**	56.26	96.01	228.46	646.73
Run time for DU	57.16	47.78	46.59	46.46	**43.71**	44.38	46.32	46.58	52.07

5.3.2 Evaluation of SL-OVS. In the following, we quantify the performance of SL-UVM and SL-OVS for the `insert()` and `search()` kernels. Both SL-UVM and SL-OVS employ batching of input keys and sorting of the keys within each batch.

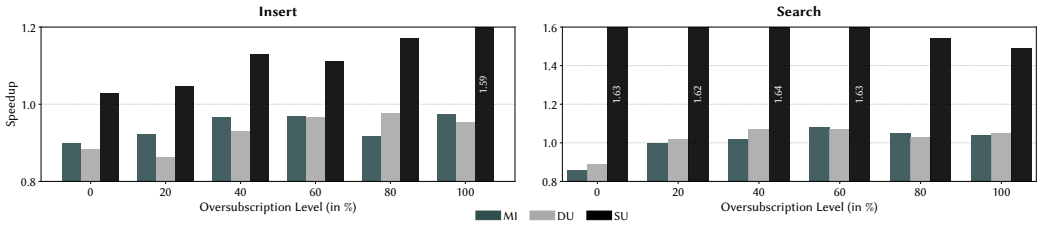


Fig. 16. Speedup of SL-OVS relative to SL-UVM for the `insert()` and `search()` kernels

Figure 16 shows the speedup achieved by SL-OVS over SL-UVM for the `insert()` and `search()` kernels. For the `insert()` kernel, SL-UVM outperforms SL-OVS on the MI and DU inputs. Inherent locality and no reuse across batches in these input sequences mute the performance benefit of SL-OVS during the build phase of the skip list. However, SL-OVS outperforms SL-UVM for the SU input. The speedup enjoyed by the SU input increases gradually with growing oversubscription levels, attaining a 1.6× speedup for 100% oversubscription. For the `search()` kernel, SL-OVS achieves a geomean speedup of 1.01×, 1.02×, and 1.59× over SL-UVM for MI, DU, and SU inputs, respectively, with a maximum speedup of 1.64× for the SU input. There are two primary reasons for this speedup. First, sorting of keys within a batch introduces spatial locality. A skip list, by design, maintains keys in a sorted order, thereby translating the locality of a sorted batch into access locality. Additionally, each batch is sized such that the pages of the skip list containing the keys within a batch fit in GPU memory. Second, the two-level architecture of SL-OVS introduces higher degrees of concurrency by allowing multiple warps to operate in parallel on different keys, each mapping to a smaller, disjoint skip list.

Sensitivity to Inner Skip List Size. A contiguous range of keys map to one inner skip list of SL-OVS. Thus, the size of each inner skip list is decided by the range size. Table 10 shows the variation in the run time of SL-OVS for the `insert()` and `search()` kernels with the MI and DU inputs at 100% memory oversubscription level as the range size is varied. The memory region reserved for each inner skip list is $256 \times (1/14)^{th}$ of the total number of elements in an inner skip list, where 256 is

the size of each super-node. With 2^{21} keys per inner skiplist in SL-OVS, the memory reserved for each inner skip list is about 36 MB. This configuration provides the best performance across both inputs for different operations.

Table 10. Sensitivity of SL-OVS to inner skip list size for the `insert()` and `search()` kernels

Range size		2^{12}	2^{15}	2^{18}	2^{21}	2^{24}	2^{27}	2^{30}
Run time (seconds)	MI (Insert)	2199.3	2239.8	2274.9	**2232.8**	2331.6	2242.5	2246.2
	MI (Search)	11.7	17.5	12.4	**11.7**	12.2	13.1	13.6
	DU (Insert)	77.3	80.4	80.0	**69.2**	120.9	90.2	89.6
	DU (Search)	26.3	28.7	18.4	**17.2**	17.27	17.4	17.4

5.4 Comparison with SUV

The SUV framework [10] aims to improve the performance of UVM-enabled applications by automatically monitoring the page access patterns and migrating appropriate pages to reduce the access latency. SUV identifies frequently-accessed memory regions based on static analysis of GPU kernel code, places them in GPU memory, and proactively evicts data that is not going to be required after a kernel ends. SUV's dynamic analysis deals with kernels with irregular accesses that are not amenable to static analysis (e.g., BFS [16]).

SUV's host code transformation passes do not support some program features such as non-UVM allocations, LLVM's ϕ-like `select` instruction, inline PTX, pointer chase patterns, and data-dependent control flow (e.g., binary search). We simplify our HT-UVM implementation to allow processing with SUV, but could not get SUV to work with SL-UVM because of errors in SUV's passes.[7] When SUV is run on a suitably modified version of HT-UVM, it fails to infer the access density and reuses in HT-UVM, and emits only a `cudaMemAdviseSetAccessedBy` hint for the auxiliary array containing the input (key, value) pairs to be operated on. With 50% memory oversubscription, our HT-UVM implementation outperforms the HT-UVM implementation with hints incorporated using SUV's static analysis by 2.17× when averaged across all input sequences. Our HT-OVS proposal would enjoy even larger benefits. These results show that SUV fails to predict the access density and reuse of these complex data structures even after simplifying the implementations to avoid transformation errors in SUV. Therefore, SUV falls back to GPU access counters for triggering run-time migration in such cases leading to poor performance.

6 Case Studies

Our results so far highlight the advantages of our design in managing and processing synthetic datasets that surpass the memory limitations of modern GPUs, demonstrating the potential for scaling real-world applications. In this section, we use our hash table and skip list designs to implement key-value stores in two real-world applications, namely, *metagenomics classification* and *k-mer counting* [35, 36]. Both applications rely heavily on key-value stores to efficiently process large-scale genomics datasets. The metagenomics classifier divides a genome into fixed substrings of length k (called a *k-mer*) and encodes each nucleotide (Adenine, Thymine, Guanine, Cytosine) of the substring using two bits to form the key K representing the substring. Each key K is associated with a *taxon_id*, which uniquely identifies an organism and acts as the value of the key. The classifier consists of two phases, namely, construction of the key-value pair database involving insert operations and the classification phase involving search operations. The k-mer counting

[7] We have confirmed the limitations of the current SUV prototype with the SUV authors.

application tracks the frequency of each k-mer in a genome. Instead of the *taxon_id*, the frequency of a k-mer serves as its value. In our evaluation, we use genome sequences from the NCBI dataset [57] for building and querying the database. We set 16 as the k-mer length in terms of the number of nucleotides leading to 32-bit keys.

Figure 17 shows the speedup of HT-OVS and SL-OVS over HT-UVM and SL-UVM respectively. For hash table, HT-OVS enjoys a speedup of up to 2.1× for metagenomics database build, 4.8× for k-mer counting and 6× for metagenomics classification phase. Even for 50% GPU memory oversubscription, HT-OVS offers speedup in the range of 1.4× to 6× over HT-UVM. SL-OVS achieves a geomean speedup of 2.4× over SL-UVM for the build phase of the metagenomics classifier and the achieved speedup increases with growing oversubscription degree. The geomean speedup achieved by SL-OVS for the metagenomics classification phase is 1.81× over SL-UVM. The k-mer counting application enjoys a geomean speedup of 6.3× with SL-OVS over SL-UVM. Overall, the performance improvement achieved by our HT-OVS and SL-OVS designs in metagenomics classification and k-mer counting establishes their utility in real-world applications.

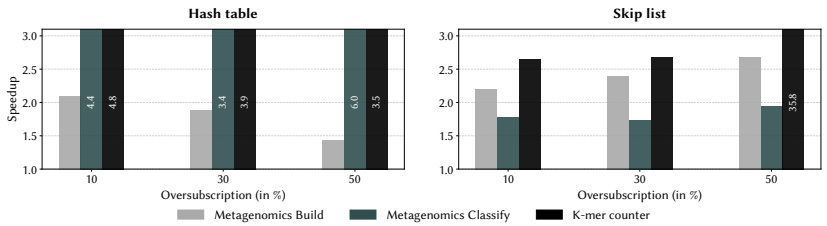


Fig. 17. Speedup achieved by HT-OVS and SL-OVS over HT-UVM and SL-UVM respectively.

7 Conclusion

In this work, we highlight the challenges in designing efficient and scalable GPU key-value stores that oversubscribe global memory. We show that it is non-trivial to identify and use memory advise and prefetch hints that can effectively address the performance drawbacks of memory oversubscription. We propose novel two-level hierarchical designs for hash table and skip list and empirically show that our proposed data structures perform better than optimized UVM implementations across a diverse set of input sequences and real-world applications. The computation pipeline involves preprocessing the input sequence for better locality, and batching the input to fit in global memory and reduce page faults. Our hierarchical two-level design paradigm paves the way for accelerating a larger class of big-data applications on discrete GPUs.

Acknowledgments

The authors thank the anonymous reviewers for their constructive feedback. We thank Pranjal Singh and Pratheek B for their help with running the SUV artifact, and thank Deven Gangwani and Shivam Sharma for their help in implementing the in-memory version of GFSL. We acknowledge the support received from the SERB Grant SRG/2019/000384, Google India Research Award 2021, research grant from Intel Corporation, and the TCS Research Scholar Program.

Data-Availability Statement

The artifact for this research is publicly available [60]. The repository includes baseline UVM implementations of hash table (i.e., HT-UVM) and skip list (i.e., SL-UVM) data structures, and our proposed designs targeting oversubscription (i.e., HT-OVS and SL-OVS).

References

[1] Umut A. Acar and Guy. E. Blelloch. 2022. *Algorithms: Parallel and Sequential*.

[2] 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. *ACM Transactions on Graphics* 28, 5 (Dec. 2009), 1–9. doi:10.1145/1618452.1618500

[3] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory. *ACM Transactions on Architecture and Code Optimization* 21, 1, Article 14 (Jan. 2024), 24 pages. doi:10.1145/3632953

[4] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *IEEE International Parallel and Distributed Processing Symposium*. 141–150. doi:10.1109/IPDPS49936.2021.00023

[5] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. Article 64, 15 pages. doi:10.1145/3458817.3480855

[6] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *IEEE International Parallel and Distributed Processing Symposium*. 419–429. doi:10.1109/IPDPS.2018.00052

[7] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 145–157. doi:10.1145/3293883.3295706

[8] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, Martín Farach-Colton, and John D. Owens. 2021. Better GPU Hash Tables. *CoRR* abs/2108.07232 (2021). doi:10.48550/ARXIV.2108.07232

[9] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2023. A GPU Multiversion B-Tree. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*. 481–493. doi:10.1145/3559009.3569681

[10] Pratheek B, Guilherme Cox, Jan Vesely, and Arkaprava Basu. 2024. SUV: Static Analysis Guided Unified Virtual Memory. In *IEEE/ACM International Symposium on Microarchitecture*. 293–308. doi:10.1109/MICRO61859.2024.00030

[11] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. 2005. Biased Skip Lists. *Algorithmica* 42, 1 (May 2005), 31–48. doi:10.1007/s00453-004-1138-6

[12] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 357–369. doi:10.1145/3018743.3018761

[13] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache your Hash on Flash. *Proceedings of the VLDB Endowment* 5, 11 (July 2012), 1627–1637. doi:10.14778/2350229.2350275

[14] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proceedings of the ACM Management of Data* 3, 3, Article 165 (June 2025), 26 pages. doi:10.1145/3725406

[15] Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. 2021. To Move or Not to Move? - Page Migration for Irregular Applications in Over-subscribed GPU Memory Systems with DynaMap. In *ACM International Conference on Systems and Storage*. Article 1, 12 pages. doi:10.1145/3456727.3463766

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*. 44–54. doi:10.1109/IISWC.2009.5306797

[17] Funda Ergun, S. Cenk Şahinalp, Jonathan Sharp, and Rakesh K. Sinha. 2001. Biased Skip Lists for Highly Skewed Access Patterns. In *Algorithm Engineering and Experimentation*. 216–229.

[18] Facebook. 2026. Folly: Facebook Open-source Library. Online. https://github.com/facebook/folly

[19] Unified Acceleration Foundation. 2025. oneAPI Threading Building Blocks (oneTBB). Online. https://github.com/uxlfoundation/oneTBB

[20] Joel Fuentes, Wei-yu Chen, Guei-yuan Lueh, and Isaac D. Scherson. 2019. A Lock-Free Skiplist for Integrated Graphics Processing Units. In *IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum*. 36–46. doi:10.1109/IPDPSW.2019.00015

[21] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *ACM/IEEE International Symposium on Computer Architecture*. 224–235. doi:10.1145/3307650.3322224

[22] Lan Gao, Yunlong Xu, Chongyang Xu, Rui Wang, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2019. Towards a General and Efficient Linked-list Hash Table on GPUs. In *IEEE International Conference on High Performance Computing and Communications*. 1452–1460. doi:10.1109/HPCC/SmartCity/DSS.2019.00201

[23] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. 2011. Coherent Parallel Hashing. *ACM Transactions on Graphics* 30, 6 (Dec. 2011), 1–8. doi:10.1145/2070781.2024195

[24] Google. 2011. leveldb. Online. https://github.com/google/leveldb
[25] Oded Green. 2021. HashGraph–Scalable Hash Tables Using a Sparse Graph Data Structure. *ACM Transactions on Parallel Computing* 8, 2, Article 11 (July 2021), 17 pages. doi:10.1145/3460872
[26] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs. *CoRR* abs/2007.09822 (2020). doi:10.48550/ARXIV.2007.09822
[27] Maurice Herlihy, Nir Shavit, Victor Luchango, and Michael Spear. 2020. *The Art of Multiprocessor Programming* (2nd ed.). Morgan Kaufmann Publishers Inc.
[28] Nathan Jones, Tyler Allen, and Rong Ge. 2025. HELM: Characterizing Unified Memory Accesses to Improve GPU Performance under Memory Oversubscription. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. 490–504. doi:10.1145/3712285.3759812
[29] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping Host-to-Device Copy and Computation Using Hidden Unified Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 321–335. doi:10.1145/3332466.3374531
[30] Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2018. WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes. In *IEEE International Parallel and Distributed Processing Symposium*. 441–450. doi:10.1109/IPDPS.2018.00054
[31] 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 *IEEE International Conference on High Performance Computing*. 11–20. doi:10.1109/HiPC50609.2020.00015
[32] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*. 63–74. doi:10.1109/PACT.2015.13
[33] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1357–1370. doi:10.1145/3373376.3378529
[34] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *USENIX Symposium on Operating Systems Design and Implementation*. 161–177.
[35] Robin Kobus, Christian Hundt, Andre Muller, and Bertil Schmidt. 2017. Accelerating Metagenomic Read Classification on CUDA-enabled GPUs. *BMC Bioinformatics* 18, Article 11 (Jan. 2017), 10 pages.
[36] Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2021. MetaCache-GPU: Ultra-Fast Metagenomic Classification. In *IEEE International Conference on Parallel Processing*. Article 25, 11 pages.
[37] Sarath Lakshman, Sriram Melkote, John Liang, and Ravi Mayuram. 2016. Nitro: A Fast, Scalable In-Memory Storage Engine for NoSQL Global Secondary Index. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1413–1424.
[38] Brenton Lessley and Hank Childs. 2020. Data-Parallel Hashing Techniques for GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 237–250.
[39] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. DyCuckoo: Dynamic Hash Tables on GPUs. In *International Conference on Data Engineering*. 744–755.
[40] Zhenxin Li, Bing Jiao, Shuibing He, and Weikuan Yu. 2022. PhaST: Hierarchical Concurrent Log-Free Skip List for Persistent Memory. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (Dec. 2022), 3929–3941.
[41] Mao Lin, Yuan Feng, Guilherme Cox, and Hyeran Jeon. 2025. Forest: Access-aware GPU UVM Management. In *ACM/IEEE International Symposium on Computer Architecture*. 137–152.
[42] Mao Lin and Hyeran Jeon. 2025. Understanding Oversubscribed Memory Management for Deep Learning Training. In *Workshop on Machine Learning and Systems*. 46–55.
[43] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! *ACM Transactions on Parallel Computing* 5, 4, Article 16 (Feb. 2019), 32 pages.
[44] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. 2023. High-Performance Filters for GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 160–173.
[45] Meta. 2012. rocksdb. Online. https://github.com/facebook/rocksdb
[46] Prabhakar Misra and Mainak Chaudhuri. 2012. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In *International Conference on Parallel and Distributed Systems*. 53–60.
[47] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. 2017. A GPU-Friendly Skiplist Algorithm. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*. 246–259.
[48] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. 1992. Deterministic Skip Lists. In *ACM-SIAM Symposium on Discrete Algorithms*. 367–375.
[49] NVIDIA. 2020. cuCollections. Online. https://github.com/NVIDIA/cuCollections
[50] NVIDIA. 2022. Boosting Application Performance with GPU Memory Prefetching. Online. https://developer.nvidia.com/blog/boosting-application-performance-with-gpu-memory-prefetching/

[51] NVIDIA. 2022. Pinterest Boosts Home Feed Engagement 16% With Switch to GPU Acceleration of Recommenders. Online. https://blogs.nvidia.com/blog/pinterest-gpu-acceleration-recommenders/
[52] NVIDIA. 2023. Maximizing Performance with Massively Parallel Hash Maps on GPUs. Online. https://developer.nvidia.com/blog/maximizing-performance-with-massively-parallel-hash-maps-on-gpus
[53] NVIDIA. 2025. Accelerated Application Catalog. Online. https://www.nvidia.com/en-us/accelerated-applications/
[54] NVIDIA. 2025. CUDA C++ Programming Guide. Online. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[55] NVIDIA Developer. 2019. CUSP. Online. https://developer.nvidia.com/cusp
[56] NVIDIA Developer. 2025. NVIDIA Nsight Systems. Online. https://docs.nvidia.com/nsight-systems/index.html
[57] Nuala A. O'Leary, Mathew W. Wright, James Rodney Brister, Stacy Ciufo, Diana Haddad, Richard McVeigh, Bhanu Rajput, Barbara Robbertse, Brian Smith-White, Danso Ako-Adjei, Alex Astashyn, Azat Badretdin, Yīmíng Bào, Olga Blinkova, Vyacheslav Brover, Vyacheslav Chetvernin, Jinna Choi, Eric Cox, Olga D. Ermolaeva, Catherine M. Farrell, Tamara Goldfarb, Tripti Gupta, Daniel H. Haft, Eneida L. Hatcher, Wratko Hlavina, Vinita S. Joardar, Vamsi K Kodali, Wenjun Li, Donna R. Maglott, Patrick Masterson, Kelly M. McGarvey, Michael R. Murphy, Kathleen O'Neill, Shashikant Pujar, Sanjida H. Rangwala, Daniel Rausch, Lillian D. Riddick, Conrad L. Schoch, Andrei Shkeda, Susan S. Storz, Hanzhen Sun, Françoise Thibaud-Nissen, Igor Tolstoy, Raymond E. Tully, Anjana Raina Vatsan, Craig Wallin, David Webb, Wendy Wu, Melissa J. Landrum, Avi Kimchi, Tatiana A. Tatusova, Michael DiCuccio, Paul A. Kitts, Terence D. Murphy, and Kim D. Pruitt. 2016. Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Research* 44, D1 (Jan 2016), D733–D745. doi:10.1093/nar/gkv1189
[58] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? *CoRR* abs/2502.10517 (2025).
[59] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Annual European Symposium on Algorithms*. 121–133.
[60] Vipin Patel, Srinjoy Sarkar, Swarnendu Biswas, and Mainak Chaudhuri. 2026. gpu-oversubscribed-ds-oopsla26. https://github.com/prospar/gpu-oversubscribed-ds-oopsla26
[61] Matt Pharr and Randima Fernando. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.
[62] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. doi:10.1145/78973.78977
[63] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *ACM Symposium on Operating Systems Principles*. 497–514. doi:10.1145/3132747.3132765
[64] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *ACM/IEEE International Symposium on Computer Architecture*. 180–192. doi:10.1109/ISCA.2018.00025
[65] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *ACM/IEEE International Symposium on Computer Architecture*. 352–363. doi:10.1109/MICRO.2018.00036
[66] Martin Steinebach. 2023. An Analysis of PhotoDNA. In *International Conference on Availability, Reliability and Security*. Article 44, 8 pages. doi:10.1145/3600160.3605048
[67] Patrick Stotko. 2019. stdgpu: Efficient STL-like Data Structures on the GPU. *CoRR* abs/1908.05936 (2019).
[68] Patrick Stotko. 2019. stdgpu: Efficient STL-like Data Structures on the GPU. Online. https://github.com/stotko/stdgpu
[69] P. Stotko, S. Krumpen, M. B. Hullin, M. Weinmann, and R. Klein. 2019. SLAMCast: Large-Scale, Real-Time 3D Reconstruction and Streaming for Immersive Multi-Client Live Telepresence. *IEEE Transactions on Visualization and Computer Graphics* 25, 5 (May 2019), 2102–2112. doi:10.1109/TVCG.2019.2899231
[70] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software*. 161–171. doi:10.1109/ISPASS.2016.7482091
[71] David B. Kirk Wen-mei W. Hwu and Izzat El Hajj. 2022. *Programming Massively Parallel Processors: A Hands-on Approach* (4th ed.). Morgan Kaufmann Publishers Inc.
[72] Lu Xing, Venkata Sai Pavan Kumar Vadrevu, and Walid G. Aref. 2025. The Ubiquitous Skiplist: A Survey of What Cannot be Skipped About the Skiplist and its Applications in Data Systems. *Comput. Surveys* 57, 11, Article 297 (June 2025), 37 pages. doi:10.1145/3736754
[73] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: a High Throughput B+tree for GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 133–144. doi:10.1145/3293883.3295704
[74] Jeseong Yeon, Leeju Kim, Youil Han, Hyeon Gyu Lee, Eunji Lee, and Bryan S. Kim. 2020. JellyFish: A Fast Skip List with MVCC. In *ACM/IFIP/USENIX International Middleware Conference*. 134–148. doi:10.1145/3423211.3425672

[75] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, and Zhiying Wang. 2020. Coordinated Page Prefetch and Eviction for Memory Oversubscription Management in GPUs. In *IEEE International Parallel and Distributed Processing Symposium*. 472–482. doi:10.1109/IPDPS47924.2020.00056

[76] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. 2020. HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2461–2474. doi:10.1109/TCAD.2019.2944790

[77] Hao Zhou, David Troendle, and Byunghyun Jang. 2021. DACHash: A Dynamic, Cache-Aware and Concurrent Hash Table on GPUs. In *IEEE International Symposium on Computer Architecture and High Performance Computing*. 1–10. doi:10.1109/SBAC-PAD53543.2021.00012

Received 2025-10-10; accepted 2026-02-17