AlignMalloc: Warp-Aware Memory Rearrangement Aligned With UVM Prefetching for Large-Scale GPU Dynamic Allocations

Jiajian Zhang, Fangyu Wu, *Member, IEEE*, Hai Jiang, *Member, IEEE*, Qiufeng Wang, *Member, IEEE*, Genlang Chen, Guangliang Cheng, Eng Gee Lim, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—As parallel computing tasks rapidly expand in both complexity and scale, the need for efficient GPU dynamic memory allocation becomes increasingly important. While progress has been made in developing dynamic allocators for substantial applications, their real-world applicability is still limited due to inefficient memory access behaviors. This paper introduces AlignMalloc, a novel memory management system that aligns with the Unified Virtual Memory (UVM) prefetching strategy, significantly enhancing both memory allocation and access performance in large-scale dynamic allocation scenarios. We analyze the fundamental inefficiencies in UVM access and first reveal the mismatch between memory access and UVM prefetching methods. To resolve this issue, AlignMalloc implements a warp-aware memory rearrangement strategy that exploits the regularity of warps to align with the UVM's static prefetching setup. Additionally, AlignMalloc introduces an OR tree-based structure within a host-co-managed framework to further optimize dynamic allocation. Comprehensive experiments demonstrate that AlignMalloc substantially outperforms current state-of-the-art systems, achieving up to $2.7\times$ improvement in dynamic allocation and $2.3\times$ in memory access. Additionally, eight real-world applications with diverse memory access patterns exhibit consistent performance enhancements, with average speedups $1.5\times$.**

***Index Terms*—Dynamic allocation, memory arrangement, unified virtual memory.**

Received 16 November 2024; revised 26 March 2025; accepted 6 May 2025. Date of publication 9 May 2025; date of current version 30 May 2025. This work was supported in part by Suzhou Science and Technology Development Planning Programme under Grant ZXL2023176, in part by the Jiangsu Province Engineering Research Centre of Data Science and Cognitive Computation at XJTLU and SIP AI Innovation Platform under Grant YZCXPT202210, in part by the Major Science and Technology Projects of Ningbo under Grant 2024Z259, and in part by Suzhou Municipal Key Laboratory for Intelligent Virtual Engineering under Grant SZS2022004. Recommended for acceptance by D. Li. *(Corresponding author: Fangyu Wu.)*

Jiajian Zhang is with the University of Liverpool, L69 3BX Liverpool, U.K., and also with the Xi'an Jiaotong-Liverpool University, Suzhou 215123, China.

Fangyu Wu, Qiufeng Wang, and Eng Gee Lim are with the Xi'an Jiaotong-Liverpool University, Suzhou 215123, China (e-mail: fangyu.wu02@xjtlu.edu.cn).

Hai Jiang is with the Beijing University of Posts and Telecommunications, Beijing 100876, China.

Genlang Chen is with the NingboTech University, Ningbo 315100, China.

Guangliang Cheng is with the University of Liverpool, L69 3BX Liverpool, U.K.

Keqin Li is with the State University of New York, New Paltz, NY 12561 USA.

Digital Object Identifier 10.1109/TPDS.2025.3568688

I. Introduction

THE advent of dynamic allocation in the CUDA Toolkit represents a significant advancement in GPU computing, simplifying memory management and eliminating the need for preallocating resources [1]. This development enhances memory flexibility and reduces the underutilization risk, allowing for more precise resource allocation tailored to applications [2], such as graph analytics [3], [4], genomics [5], and sparse linear computing [6]. As parallel GPU computing expands, large-scale dynamic allocation has increasingly garnered the attention of researchers [7], [8], [9], [10]. However, the limitations of CUDA's *malloc* at large scales are characterized by significant synchronization overhead [11] and suboptimal memory utilization that rarely exceeds 40% [12]. Consequently, these challenges compel applications that require efficient allocation with extensive data to revert to static memory pre-allocation [7], [8], [13].

Current research on GPU dynamic memory allocation [1], similar to CUDA's *malloc*, also faces challenges in supporting scalable scenarios. Our experiments reveal that the dynamic allocated memory of state-of-the-art (SOTA) systems, such as Ouroboros [14] and Synchronization allocator [15], are constrained to dynamic memory allocations of 2 GB, which is only about 10% of the physical memory capacity of the devices. This limitation primarily arises from the inherent design of GPU memory management systems, which rely heavily on GPU computing units and storage for all memory operations, often leading to rapid resource depletion. Gallatin [16], the first to propose a 3-level van Emde Boas (vEB) tree [17] for supporting large-scale allocations, still encounters practical difficulties due to reliance on 64-bit GPU atomic operations and static memory data structures. To overcome this limitation, SyncMalloc [12] introduces a host-GPU collaborative management framework that integrates with the GPU's UVM [18] for large-scale dynamic allocation, rather than solely coupling with the GPU devices themselves.

However, current research on GPU dynamic allocation [1], [12], [16] primarily focuses on the performance of the allocation itself, while often overlooking the alignment with the GPU's memory access behaviors. In practical applications, memory is allocated just once but accessed multiple times [19]. Ignoring the performance implications of these repetitive memory accesses can compromise their effectiveness. For instance, the discrete

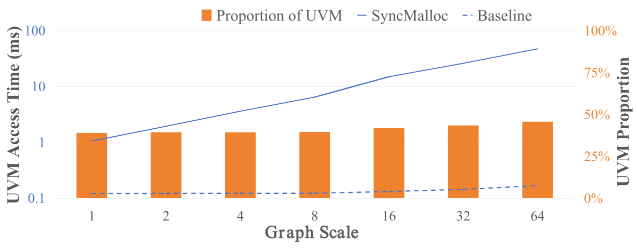


Fig. 1. Memory Access Performance of Random Walk.

memory blocks designed in Synchronization allocator [15] can lead to decreased memory coalescing, impacting the overall memory access efficiency, despite the satisfactory allocation performance. Moreover, mismatches of memory access behaviors in UVM systems introduce more significant overhead. Different from accesses within the GPU, which may result in *L1* or *L2* cache misses, UVM mismatches often lead to frequent page transfers between the host and the device, causing more time costs. Through a graph clustering test with random walk [8], [20], we further discover significant inefficiencies in UVM access within the current SOTA system [12]. As shown in Fig. 1, SyncMalloc's cost is up to 200$\times$ higher than the baseline scenario focusing solely on random access time [20], with UVM-related costs accounting for half of the total runtime as the graph scale increases.

To address the challenges of UVM mismatches in GPU memory management, we propose AlignMalloc, a novel dynamic allocation system specifically designed to align memory arrangement with the underlying UVM prefetching strategy for large-scale applications. By leveraging the unique characteristics of GPU warps, warp-aware memory rearrangement is introduced as a strategic guide to optimize memory layouts during dynamic memory allocation. AlignMalloc provides users with specific libraries for allocation and access at the application level. To the best of our knowledge, AlignMalloc is the first system to comprehensively address both memory allocation and access performance concurrently.

AlignMalloc is designed to effectively tackle the inefficiencies associated with UVM memory access. In our experimental analysis of a graph clustering scenario, we broaden our understanding by identifying a correlation between thread access sparsity and the efficiency of UVM prefetching. Further exploration into tracking page faults within the UVM strategy highlights a **fundamental issue**: the dynamic nature of memory access, which is dictated by the running program's logic, contrasts with the static setup of the UVM scheduling and scopes. This mismatch leads to diminished prefetching hit rates and the increased prefetching pages, thereby incurring the associated overheads.

To tackle the above foundational discrepancy, AlignMalloc introduces a warp-aware memory rearrangement that aligns closely with the underlying UVM prefetching strategy for large-scale scenarios. Recognizing the homogeneity in memory access behaviors among threads within a warp, we leverage the regularity introduced by warps to act as a bridge between the static setup of UVM prefetching and actual memory arrangement, ensuring consistency. Specifically, we transpose the memory layout focusing from on individual threads to the warp-level behavior, enhancing the prefetching hit rate by coordinating with UVM prefetching scheduling. Moreover, to further align with the UVM prefetching scope, AlignMalloc introduces a subtree-based arrangement that organizes memory across warps, which is designed to minimize prefetching interference among different warps. These adjustments at different granularity levels—within individual warps (intra-warp) and among multiple warps (inter-warp) phases—constitute the core of our warp-aware rearrangement strategy.

Building on the warp-aware rearrangement strategy, we introduce a host-co-managed memory system within an OR tree-based structure to enhance the management of dynamic allocation. This architecture processes memory requests organized by the GPU warp units to align with warp-aware rearrangement. Each node of the OR tree recursively stores a logical OR value, indicating the availability of memory blocks involved by its child leaf nodes. It allows for an operational time complexity of $O(\log n)$ for the memory allocation without requiring a large amount of memory space.

Our contributions can be summarized as follows:

- We propose AlignMalloc, a solution that directly addresses the fundamental issue of inefficiency in UVM prefetching, significantly enhancing both memory allocation and access performance for large-scale applications.
- AlignMalloc introduces a warp-aware memory rearrangement, which leverages the regularity and locality inherent in the warp to align with the UVM static strategies.
- A memory management system within an OR tree-based architecture co-managed with the host is introduced in AlignMalloc, to enhance the management of dynamic allocation with a warp-aware rearrangement strategy.
- Our proposed system, evaluated through comprehensive experiments, significantly outperforms current SOTA dynamic allocators in both allocation and memory access performance within large-scale scenarios.

II. Background

A. Page Fault Processing in UVM

Unified Virtual Memory (UVM) was introduced in CUDA 6.0 as a paradigm to unify the address spaces of CPUs and GPUs [21], simplifying memory management in heterogeneous computing environments. UVM establishes a single unified address space, which facilitates seamless memory access for both host and device kernels under the same virtual environment, without the need for explicit management of memory-specific locations. This abstraction allows programmers to focus on tasks themselves rather than the intricacies of host-device memory migration, as the UVM driver handles the scheduling of memory resources across devices.

When a memory resource accessed by a warp is not present on the local device, a page fault is triggered, causing the warp to stall while waiting for the remote resource to be fetched [22]. During this waiting period, the warp scheduler proactively swaps in other non-faulting warps, effectively masking the memory access latency and maintaining overall device efficiency. To further enhance the efficiency of host-device transfers during fault processing, GPUs employ a batching process for handling faults. This process involves collecting multiple faults in a page fault buffer until a predetermined threshold is reached. Once accumulated, the batch of faults is serviced by the host, which allocates the appropriate physical pages and transfers them back to the requesting device. Following the processing, a replay signal is sent to the device to wake up the warps and the GPU is instructed to flush the page fault buffer and prepare for the next iteration of fault handling.

B. Prefetching Strategy

To mitigate the substantial overhead associated with fault processing and the subsequent stalling of warps, UVM has introduced a strategy known as prefetching, which involves fetching data in advance to the device, effectively reducing the frequency of page faults [23]. UVM prefetching can be divided into two types. The **first type** allows programmers to provide hints about UVM memory blocks, including *preferred locations*, *read mostly settings*, and *access with specific devices*. By leveraging these hints, UVM devises appropriate scheduling strategies for the memory blocks. However, this prefetching approach requires programmers to possess a high degree of foresight regarding the usage patterns of memory blocks.

The **second type** of prefetching, is dynamically managed at runtime by the underlying UVM scheduler. This strategy uses a 64 KB block as the basic unit for prefetching rather than a standard page. Specifically, during fault processing, the UVM system assesses data locality by examining the physical locations of blocks adjacent to the one experiencing the fault. This locality assessment leverages the Tree-based Neighborhood (TBN) data structure to predict which memory blocks will likely be needed shortly and fetches them to the device alongside the faulted block. Our paper primarily focuses on optimizing the memory arrangement to align with this dynamic prefetching mechanism, aiming to significantly improve the hit rate of prefetching.

C. Tree-Based Neighborhood Structure

The TBN is a full binary tree comprising 63 nodes, including 32 leaf nodes, each representing a basic block within a continuous segment of UVM address space. Capable of representing up to 2 MB of UVM memory, termed a VABlock, the prefetching scope is confined to this 2 MB segment.

Fig. 2 simplifies this semantics with a 15-node TBN illustration. The capacity of each tree node reflects the ratio of basic blocks that are physically present in the device, as covered by the node's child leaf nodes. In Fig. 2, $Node_1$'s capacity hits 50%, indicating that two ($Blocks_{1,3}$) out of the four basic blocks represented by its child leaf nodes ($Blocks_{0-3}$) are located in the target device. When a fault occurs, the node capacities are updated recursively from the affected leaf node up to the root. If any node's capacity surpasses the set maximum threshold (default set at 50%), the basic blocks corresponding to that node's child leaf nodes are transferred to the device as part of the prefetching operation. For example, in Fig. 2, should a

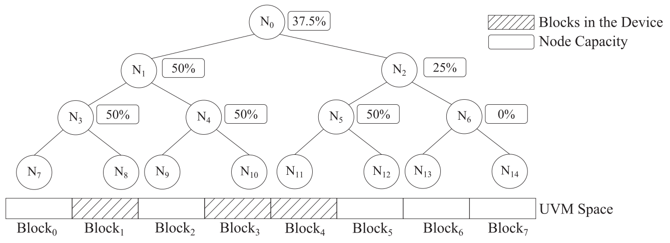


Fig. 2. Demonstration of TBN with 15 Nodes.

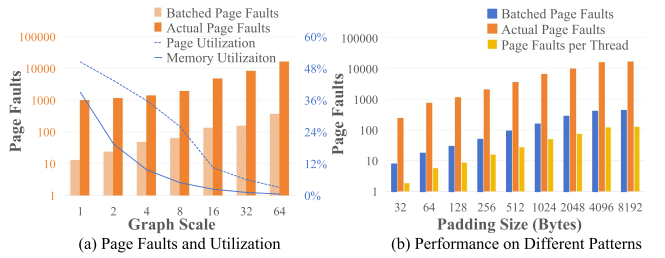


Fig. 3. Details of UVM Page Faults. (a) Correlation between page faults and graph scales. (b) Relationship between page faults and the sparsity of memory access.

fault in $Block_0$ trigger, $Node_1$'s capacity would update to 75%, exceeding the threshold. Subsequently, $Block_2$, too, would be prefetched to the device.

III. Motivation

This section motivates the proposed warp-aware rearrangement by analyzing the foundational mismatch between the memory arrangement and UVM prefetching.

A. A Random Walk Case

In this setup, 1,000 threads perform a random walk on a graph structure as an adjacency list, dynamically allocating their linked lists within the UVM space. We vary the graph's scale by adjusting node counts and connectivity within the graph. For instance, the graph scale 1 is defined by setting the number of nodes at 16 K and the size of each thread's adjacency list at 128 edges. Fig. 1 highlights the significant overhead caused by UVM in graph clustering using a dataset [24], [25] commonly employed by advanced dynamic allocators [1], [16]. To gain deeper insights, we use page faults and prefetching effectiveness as key metrics [26]. Fig. 3(a) shows that increased page faults directly correlate with higher UVM access overhead. Furthermore, as the scale of the graph expands, a significant decrease in utilization is observed, indicating many pages remain unused. The evaluation of the random walk case reveals that UVM is inefficient in managing large-scale graph data.

Theoretically, when the scale of access—specifically, the quantity of edges and nodes accessed—remains constant, the amount of memory introduced from the host to the GPU should also remain stable, regardless of changes in the graph's scale. However, experimental results indicate that an increase in the graph's scale leads to a corresponding increase in the number

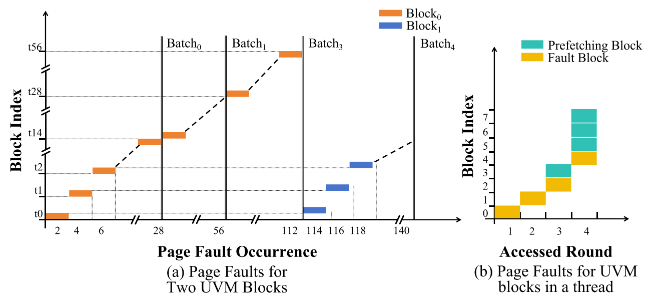


Fig. 4. Tracking Detailed Fault Processing in Designated Access Blocks.

of UVM pages and overall memory usage. We delve into the memory access patterns of the application to understand this phenomenon better. A random walk of a thread involves accessing a linked list from the graph's adjacency list, sequentially organized in the UVM. The memory allocated to $thread_{i+1}$ immediately follows the address allocated to $thread_i$. As the graph expands, the intervals between the memory spaces accessed by consecutive threads widen. Based on this observation, we hypothesize that the enlarged intervals which lead each thread to access broader memory spaces, may incur more irrelevant pages into the UVM and thereby generating significant overhead.

B. General Access Pattern Evaluation

To validate the hypothesis proposed in Section III-A, we extend our investigation to more general problems by manipulating the intervals between memory spaces of consecutive threads. By setting the different padding sizes in the memory intervals, we elucidate the correlation between the sparsity of memory intervals and UVM prefetching. Results depicted in Fig. 3(b) illustrate that the number of page faults increases with the node padding size; however, this trend begins to plateau after a padding of 4 KB, eventually stabilizing. Specifically, at a padding of 4 KB, that is a 2 MB gap between thread accesses, the number of page faults reached an average of 120 per thread. Compared to an average of 50 random accesses per thread, half of the prefetching activity is wasted on unused memory blocks, highlighting a high volume of irrelevant prefetching faults. However, when the padding is reduced to $64B$ or $32B$, the page faults drop significantly to about 5 per thread, demonstrating a 10 : 1 memory reuse ratio for prefetching. These findings underscore that different accessed memory sparsity yields markedly different impacts on prefetching efficiency. The sparser the thread access, the greater volume of irrelevant memory introduced by prefetching.

C. Analysis on UVM Strategy

To investigate how the UVM prefetching strategy fails to accurately prefetch relevant pages in sparse memory access scenarios, we track page faults across designated memory blocks. Each thread is assigned to access blocks indexed by 0, 1, 2, and 4, each block sized at 64 KB.

Fig. 4(a) shows the occurrence of memory faults when all threads access $block_{0,1}$ in parallel. Specifically, in $batch_0$, the page fault buffer collects page faults from $block_0$ for 14 threads in $warp_0$ and dispatches them to the host for processing. This procedure is repeated in two subsequent batches for the remaining faults within $warp_0$. The simultaneous triggering of page faults by the threads within a warp enables the UVM strategy to process similar memory patterns in batches, which enhances the optimization of UVM page faults through improved memory coalescing.

However, Fig. 4(b) reveals prefetching inefficiencies by focusing on page fault specifics and subsequent prefetching activities within a thread. At the end of the $blocks_{0,1}$ accesses, the prefetcher erroneously brings in $block_3$ during the processing of $block_2$'s faults and imports $blocks_{5-7}$ when accessing $block_4$, despite no actual access requirements in our access plan. This misprediction stems from the UVM prefetching strategy, which predicts future access by determining if memory blocks in the contiguous UVM context have been migrated to the GPU. Given that $blocks_{0-4}$ are already loaded, the strategy assumes a high likelihood of subsequent accesses to $blocks_{5-7}$, prompting their premature migration. However, the actual access by threads to these blocks is highly uncertain due to the block allocation within the same thread space, introducing potential access misses, magnified across threads sharing similar access logic. Nonetheless, reducing the memory intervals can substantially alter outcomes. Since the UVM prefetching strategy remains constant, the previously unaccessed $blocks_{5-7}$ may be accessed by other threads, potentially enhancing the prefetching hit rate. Thus, although batch processing in UVM enhances memory coalescing in page faulting, the uncertainty in thread access significantly impacts the effectiveness of UVM prefetching, which arises from the dynamic nature of thread parallel access.

D. Fundamental Causes

The UVM prefetching strategy, initially modeled after traditional CPU memory management practices [27], evaluates data locality by examining the physical locations of memory blocks adjacent to the fault block within the UVM. However, this strategy faces challenges within GPU applications where memory access patterns are dynamically determined by the program at runtime. This dynamic nature of parallel execution often misaligns with the relatively static setup of UVM prefetching strategies and scopes.

For instance, as depicted in Fig. 5(a), in scenarios involving large-scale data, each thread may manage multiple basic blocks. If a page fault occurs at $Block_1$, the UVM prefetching strategy will decide on prefetching based on whether the adjacent $Blocks_{0,2}$ are within the target device, but these UVM blocks still pertains solely to $thread_0$'s space. Essentially, UVM prefetching analyses the memory usage within $Thread_0$ in a one-dimensional, localized manner similar to CPU analysis for single process. This strategy, while straightforward, could lead to erroneous prefetching predictions. In the CPU context, such errors might not significantly impact performance due to the serial operation in a single process. However, in GPUs, utilizing the Single Instruction, Multiple Threads (SIMT) architecture, an incorrect prefetching prediction at single instruction could

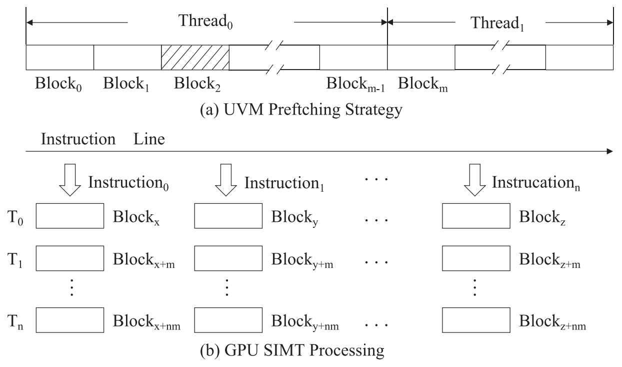


Fig. 5. Mismatch between the UVM Prefetching Strategy and GPU Processing.

lead to compounded errors across multiple threads due to similar memory access behaviors, thereby exacerbating prefetching inefficiencies. This raises the question of whether UVM prefetching in GPUs could be restructured to predict the data locality across a group of threads simultaneously at a single instruction point. For example, as depicted in Fig. 5(b), by predicting access to $block_{x+i\times m}$ for all threads at $Instruction_0$, prefetching could not only be made more accurate but also prevent the amplification of single-thread prediction errors across concurrent threads.

IV. Related Work

A. GPU Dynamic Allocation

Dynamic memory management structures for GPU applications are generally categorized into three types: array, linked list, and hybrid. Halloc [28] first introduced an array-based system that segmented memory into pre-divided fixed-size pages and utilized arrays to manage the states of these pages. To mitigate synchronization overhead and prevent race conditions amid massive parallel memory requests, Halloc employed hash functions to reduce the frequency of request collisions. Building on this structure, Pham et al. [15] proposed a Synchronization Allocator that implemented threads to randomly search for free pages, applying memory locks at the discrete page level rather than the broader management operation level, thus minimizing the granularity of synchronization. To address the significant fragmentation caused by static pre-division in array-based systems, Throughput-Oriented [29] and NBBS [30], employed linked lists to manage memory segments of flexible sizes, enhancing adaptability to varying request sizes. Specifically, these systems progressively subdivided the entire memory space into smaller buddy blocks until the size met the memory request, thereby optimizing the utilization of available memory. However, maintaining the coherence of linked lists in parallel setups can incur substantial overhead.

Drawing inspiration from the Hoard's [31] multi-level strategy for CPU, hybrid frameworks like XMalloc [32], ScatterAlloc [33], and FDG [34] combined the strengths of both linked list and array-based structures. These systems managed large memory allocations using linked lists and governed smaller allocations with arrays, striking a balance that minimized both overhead and fragmentation. To further accommodate diverse sizes of allocation requests, Ouroboros [14] introduced additional hierarchical levels, including virtualized queues, chunks, and pages.

Different from conventional structures, Gallatin [16] introduced vEB trees, designed to support allocation operations with a complexity of $O(\log\, \log\, n)$. However, due to the limited atomic capacity of GPUs, Gallatin's practical performance often suffered a substantial loss. Additionally, the vEB tree structure consumed considerable memory resources. In contrast, AlignMalloc employed a novel OR tree-based management structure that efficiently handled large memory requests within warp without requiring extensive memory space. Furthermore, this structure seamlessly integrated with the warp-aware memory rearrangement, providing a solid foundation for optimizing memory access.

B. UVM Prefetching

While existing research on UVM prefetching optimization acknowledges the issue of overly aggressive prefetching [22], it has not delved into the underlying causes as comprehensively as our study. Typically, current research solutions focus on dynamically adjusting the prefetching based on real-time observations of the GPU environment.

Building on a detailed analysis of the TBN structure in UVM prefetching strategy [23], Ganguly et al. [35] proposed an online placement strategy that dynamically adjusted the threshold for page placement based on memory patterns in CPU-GPU interconnect traffic. To enhance real-time performance and adapt to seasonal information trends, memory utilization [36] and historical page fault records [37] were incorporated to guide dynamic modifications in prefetching. Concurrently, Yu et al. [38] proposed a cost model based on these factors and employed statistical analysis to fine-tune the intensity of prefetching. Alternatively, Li et al. [39] employed throttling of GPU SMs to modulate prefetching intensity in response to excessive page faults. Addressing the limitations of throttling in large-scale scenarios, Kim et al. [40] increased thread concurrency by dispatching additional thread blocks to an SM. However, merely reducing prefetching intensity failed to address the underlying problem—the imprecise perception of locality leading to low prefetching accuracy. Furthermore, Nvidia's latest UVM driver release [41] explicitly acknowledged a bug related to the complex TBN, which prevented support for dynamic threshold modifications for prefetching.

Different from existing research, AlignMalloc optimizes memory arrangements during allocation to align with the UVM prefetching strategy. By preserving the integrity of the underlying UVM strategy, AlignMalloc mitigates potential disruptions to other applications.

V. Design of AlignMalloc

To address the fundamental discrepancy in UVM access, AlignMalloc introduces a warp-aware memory rearrangement strategy that aligns the memory access layout with the underlying UVM prefetching strategy for large-scale dynamic allocation

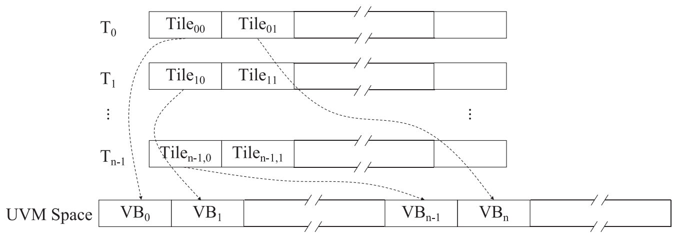


Fig. 6. Memory Pattern Transpose.

applications. Warps, serving as logical units in GPU, provide a stable basis for predicting memory access due to the consistent access behavior exhibited by the threads within a single warp, making them ideal intermediaries to bridge the gap between the dynamic memory access arrangements and the static setup of UVM prefetching. By shifting the focus from individual threads to whole warps, the strategy can leverage the inherent regularity in warp behavior, significantly enhancing the accuracy of prefetching by achieving memory coalescence.

The warp-aware strategy consists of two phases, each addressing different granularity levels. The intra-warp rearrangement is fine-grained, focusing on rearranging the memory layout of individual threads within a warp to align closely with the UVM scheduling units. Such alignment enables the UVM strategy to accurately detect and respond to the thread access similarity within a warp. For the inter-warp rearrangement, memory allocated to a warp is aligned with the UVM prefetching scope, effectively synchronizing the execution logic partitions with UVM's overall strategy and scope. This alignment ensures that prefetching within each warp remains independent, preventing interference from other warps.

A. Intra-Warp Rearrangement

During the intra-warp rearrangement phase, we refine the memory layout within each warp to align with UVM scheduling units, enhancing prefetching accuracy. As shown in Fig. 6, the dynamic memory requests from threads t_0 through t_{n-1} within a warp are divided into tiles according to the UVM's scheduling unit of 64 KB. Any remaining portions that do not meet this size are further subdivided into segments of $2^i \times 4$ KB, $i \in Z$. The system distributes tiles of each thread into Virtual Blocks (VBs) alternately within the UVM space, rearranging the memory layout with fine granularity to align with prefetching units. While the existing UVM prefetching strategy, which predicts the memory locality based on its contexts in the GPU device, remains unchanged, the UVM memory layout is adjusted to better match this strategy. When a thread accesses a memory block in UVM, the adjacent memory blocks accessed by other threads within the same warp tend to exhibit highly similar access behaviors. Thus, if a memory block triggers a page fault, the adjacent blocks are likely to be accessed soon, taking advantage of the regularity in warp behavior. This rearrangement shifts the UVM prefetching from predicting the access behavior of individual threads to anticipating the collective behavior of multiple threads within a warp, which achieves memory coalescence within warps, consequently enhancing the accuracy of prefetch predictions.

In actual UVM scheduling, page faults are handled in batches rather than individually. This batch processing may split page faults within a warp, potentially causing deviations from the warp-aware scheduling. To maximize prefetching, we optimize the memory access pattern of threads within a warp. For instance, the initial part of the pattern is arranged in the sequence thread of 0, 1, 2, 4, 6, etc., where prefetching is most likely to be activated. The sequence thread of 3, 5, 7, and so forth is placed towards the end of the pattern as targets for future prefetching. Additionally, at the end of the batch, we ensure the pattern remains sufficiently discrete to provoke prefetching for the next batch.

AlignMalloc employs a systematic address translation to map user thread's memory access ($User_{addr}$) to the corresponding UVM address (UVM_{addr}). When memory is accessed via the AlignMalloc-specific library, $User_{addr}$ is passed as a parameter to retrieve the actual address. The translation starts by utilizing the base address of the dynamic memory space to establish the initial offset (OF). This offset serves as a critical step in determining the block offset (OB), which identifies the specific block containing the data. (OB) is calculated as follows:

$$OB = OF/64\text{ KB} \tag{1}$$

Following this, the translation to the actual UVM address is computed using a specific formula:

$$UVM_{addr} = User_{addr} + (OB \times WS + Lane_{ID} - OB) \times 64 \tag{2}$$

where WS represents the number of threads in the current warp, and $Lane_{ID}$ denotes the thread ID within that warp.

B. Inter-Warp Rearrangement

When the memory layout across warps is densely allocated and controlled by a single prefetching unit, the warp prefetching strategy may be adversely affected by adjacent warps, potentially introducing irrelevant pages from them. For instance, consider the memory distribution depicted in Fig. 2, where the memory managed by subtree rooted at N_1 represents the tail end of $warp_0$'s memory blocks, while subtree N_2 represents the beginning portion of memory for $warp_1$, both within the same VABlock. When a page fault triggered by $block_0$ occurs, and because N_1's capacity exceeds its maximum threshold, $block_2$ is prefetched to the device. Concurrently, because the combined capacity of N_0 exceeds 50%, it triggers the prefetching of all blocks ($blocks_{5-7}$) under N_2, which are actually part of $warp_1$. Although prefetching $block_2$ aligns with the access patterns of $warp_0$, dragging blocks from $warp_1$ into the prefetch operation due to the mismatch of the actual usage patterns can lead to irrelevant data being fetched.

To minimize this cross-warp interference in prefetching decisions, AlignMalloc employs an inter-warp rearrangement that precisely aligns the memory allocated to each warp with the UVM prefetching scope. Specifically, our method leverages a dynamic subtree-based strategy that utilizes padding within the unified virtual space. This padding serves to distinctly delineate the prefetching boundaries for each warp, ensuring that the

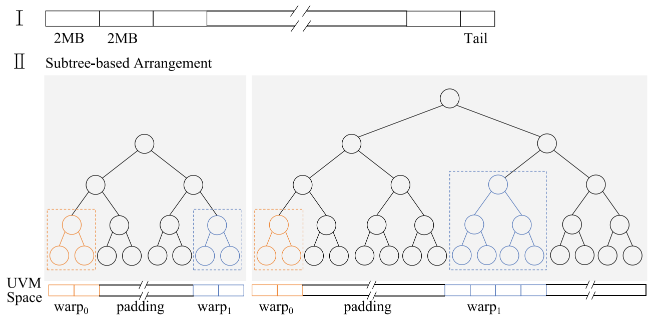


Fig. 7. Inter-warp Rearrangement.

Algorithm 1: Initialization of PortionSpace.

```
1: Create *PortionSpace* with VABlocks
2: Build *PortionTree* as a binary full tree
3: **for** each node $n$ in *PortionTree* **do**
4:     $FreeNodeArray[n] \leftarrow 1$
5: **end for**
```

Algorithm 2: Memory Allocation in PortionTree.

```
Require: Requested size $M_p$
1: $S \leftarrow 2^{\lceil \log_2(M_p/64\ \text{KB}) \rceil} \times 64$ KB
2: $h \leftarrow \lceil \log_2(M_p/64\ \text{KB}) \rceil + 1$
3: $retAddr \leftarrow -1$
4: **for** each node $n$ at height $h$ **do**
5:   **if** $FreeNodeArray[n] == 1$ **then**
6:     Allocate memory to a leaf node under subtree $n$
7:     $retAddr \leftarrow$ address of $FreeNodeArray[n]$
8:     /* update node status */
9:     $p \leftarrow$ leaf node index allocated
10:    **while** $p$ is not root **do**
11:      $p \leftarrow$ parent of $p$
12:      **if** any child node of $p$ is marked 0 **then**
13:        $FreeNodeArray[p] \leftarrow 0$
14:      **end if**
15:    **end while**
16:    **break**
17:  **end if**
18: **end for**
19: **return** $retAddr$
```

UVM prefetching scheduling is concentrated exclusively on the memory segments being accessed by the current warp, thus avoiding influence from other warps. Additionally, this strategy ensures these paddings are primarily retained within the virtual space and not translated into physical memory space during prefetching.

Fig. 7 delineates how the warp memory is organized with the size, M_w. When M_w exceeds the VABlock size (2 MB), the allocated memory of a warp is divided into 2 MB blocks, as shown in Fig. 7(I). For any portions under 2 MB, Fig. 7(II) illustrates the subtree-based arrangement that consolidates these segments into a VABlock's space. Specifically, a warp space sized $2^i \times 64$ KB is structured for M_w, where $2^{(i-1)} \times 64\ \text{KB} < M_w \leq 2^i \times 64$ KB, $i \in Z$. Within a VABlock, a binary full subtree of height $i+1$, matching the leaf node configuration of TBN, is allocated for this warp space, named $warptree$. The allocation of these subtrees must satisfy that the binary full trees formed by the parents of the root nodes of any two adjacent $warptrees$ within a VABlock do not overlap. Fig. 7 exemplifies the arrangement of two warp subtrees, and the respective formula is defined as follows:

$$\forall T_1,\ T_2 \in V_t,\ T_1' \cap T_2' = \emptyset$$
$$\text{where } r(T_1') = p(r(T_1)),\ r\left(T_2'\right) = p\left(r(T_2)\right) \quad (3)$$

where V_t represents the set of $warptrees$ within a VABlock space, and p and r denote the functions to get the parent node and root node, respectively.

Portions smaller than 2 MB are arranged using a subtree-based algorithm within a dedicated UVM space, *PortionSpace*. PortionSpace consists of 2048 VABlocks and is managed by a binary full tree structure, *PortionTree*, based on the TBN model, as shown in Algorithm 1. A bit array, *FreeNodeArray*, is employed to represent the free status of nodes in PortionTree. If a binary subtree rooted at a node has all its leaf nodes available (i.e., unallocated), the corresponding element is marked as '1'; otherwise, it is marked as '0'. When allocating a portion of size M_p, the portion is expanded to the nearest size that fits a binary subtree, $2^{\lceil \log_2(\frac{M_p}{64\ \text{KB}}) \rceil} \times 64$ KB, forming a $warptree$. The subtree-based algorithm, shown in Algorithm 2, searches FreeNodeArray to identify a free node at the height of the parent node of the root, $\lceil \log_2(\frac{M_p}{64\ \text{KB}}) \rceil + 1$. Once a free node is located, the portion is allocated to the corresponding leaf node in PortionTree. Following the allocation, the node status is recursively updated from the leaf node upwards to the root.

Before demonstrating how our subtree-based arrangement effectively prevents interference from other warps during prefetching processes, we delve into the underlying prefetching strategy with TBN and further propose several theorems in the relationship between memory occupancy and prefetching dynamics. In this framework, T represents TBN nodes of a VABlock, except for the leaf nodes. Functions p, b, and c are defined to identify the parent, sibling, and children nodes, respectively. Furthermore, U is employed to quantify the capacity of each node, specifically measuring the proportion of UVM blocks that are currently loaded into the GPU's physical memory. The maximum capacity threshold is maintained at the default setting of 50%.

Theorem 1: $\forall t \in T, 0 \leq U(t) \leq 50\%$, or $U(t) = 100\%$.

Proof: Consider the prefetching strategy, which indicates once $U(t) > 50\%$, all memory blocks associated with t are migrated to GPU devices, resulting in $u(t) = 100\%$. ∎

Theorem 2: $\forall t \in T$, if a prefetching occurs at t, then either $U(b(t))$, or $U(c(t))$ updates.

Proof: In the architecture of the UVM system as disclosed by NVIDIA [41], the updating U strategy is inherently recursive, originating from the leaf nodes during page faults. $\forall t \in T$, the update to $U(t)$ is triggered by one of two primary events. The first is an update to $U(c(t))$, which subsequently propels a recursive update upward through the node's ancestry. The second trigger occurs when $U(b(t))$ is updated, and the $U(p(t)) > \delta$, a predetermined prefetching threshold. This condition initiates prefetching activities, leading the UVM system to migrate all memory blocks associated with the child leaf nodes of $p(t)$ to the GPU's physical memory. ■

Theorem 3: $\forall t \in T$, if a prefetching occurs at t for the update of $U(b(t))$, then $U(b(t)) = 100\%$, and $U(t) > 0$.

Proof: $\forall t \in T$, a prefetching occurring at t for the update of $U(b(t))$ implies that $u(p(t)) > \delta$, the prefetching threshold.

Let $f(t) = U(p(t)) - \delta = \frac{U(b(t))+U(t)}{2} - \delta$. For $f(x) > 0$, it is necessary that $U(t) > 0$, when $\delta \geq 50\%$; or $U(t) > \frac{\delta - 50\%}{2}$, when $\delta < 50\%$. Now, $\delta = 50\%$, then $U(t) > 0$.

Suppose, that $\exists t'$, $U(b(t')) \neq 100\%$. According to Theorem 1, this implies $U(b(t')) \leq \delta$. Consequently, $U(p(t')) = \frac{U(b(t'))+U(t')}{2} \leq \delta$. This is a contradiction because $\forall t' \in T, U(p(t')) > \delta$. ■

Demonstrating the effectiveness of the subtree-based method in preventing interference among warps: This proof is structured into two parts. Firstly, we prove the necessity for the nodes formed by the warp memory blocks to be organized as a binary full subtree. Secondly, we address the specific conditions necessary for arranging the root nodes of the $warptrees$ within a VABlock to remain independent.

Proof of the Necessity for N_w to be a Binary Full Subtree: $\forall n \in N_w$, n is not interfered by prefetching from other warps. This isolation implies $\forall n'$ that can affect prefetching, then $n' \in N_w$. According to Theorem 2, this results $\forall n \in N_w$, $b(n) \in N_w$, and $c(n) \in N_w$, thereby implying $CI(N_w) = N_w$. $N_w \subset T$, then N_w is an induced subtree of T. ■

Proof of Arrangement: N_w is a binary full tree. $root(N_w)$ is not interfered with other warps. This implies $\nexists b(root(N_w))$, or according to Theorem 3, $b(root(N_w)) = 0$. *(i)* If $\nexists b(root(N_w))$, then N_w is T. *(ii)* If $b(root(N_w)) = 0$, then the binary full trees formed by $p(root(N_w))$ of any two adjacent $warptrees$ do not overlap, according to Theorem 2. ■

In the subtree-based arrangement, the memory intervals between $warptrees$ are designated as padding. Our proof demonstrates that memory migration induced by prefetching is confined exclusively to the $warptree$. As a result, the padding areas outside of these subtrees do not undergo migration to the GPU, thereby, not occupying significant GPU physical memory space. Additionally, because prefetching activities are limited to the subtree, any updates to the memory utilization U triggered by page faults are confined to this area as well. This localization restricts the recursive updating of U to the path from the leaf to the root node of the subtree, without involving any of the padding nodes. Thus, it also introduces minimal computational overhead during the prefetching process.

VI. Memory Management System

A. Overview

To enhance the management of dynamic memory allocation within the UVM system, the proposed AlignMalloc integrates the warp-aware rearrangement strategy into a host-device co-managed memory system. Fig. 8 illustrates the architecture of this system, which is divided into three main components: the host side, the device side, and the UVM space. The UVM space serves not only for dynamic allocation but also acts as a conduit for communication between the host and device. The memory management system is designed to optimize the handling of memory requests at two distinct levels of granularity: a fine-grained level within warps in GPU devices and a coarse-grained level managed by the host, each specifically designed to support the unique needs of warp-aware rearrangement.

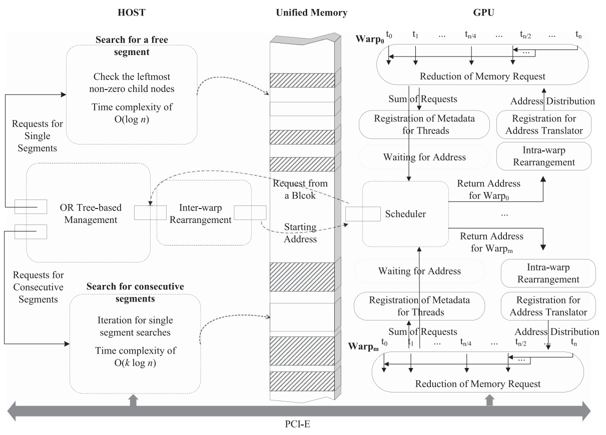

HOST
Unified Memory
GPU
Search for a free segment
Check the leftmost non-zero child nodes
Time complexity of O(log n)
Requests for Single Segments
OR Tree-based Management
Inter-warp Rearrangement
Requests for Consecutive Segments
Search for consecutive segments
Iteration for single segment searches
Time complexity of O(k log n)
Request from a Blcok
Starting Address
Warp0 t0 t1 ... tn/4 ... tn/2 ... tn
Reduction of Memory Request
Sum of Requests
Address Distribution
Registration of Metadata for Threads
Registration for Address Translator
Waiting for Address
Intra-warp Rearrangement
Scheduler
Return Address for Warp0
...
Return Address for Warpm
Waiting for Address
Intra-warp Rearrangement
Registration of Metadata for Threads
Registration for Address Translator
Sum of Requests
Address Distribution
Warpm t0 t1 ... tn/4 ... tn/2 ... tn
Reduction of Memory Request
PCI-E

Fig. 8. The Architecture of Memory Management.

At the fine level, the Warp Scheduler consolidates thread allocation requests within a warp into a unified one via a reduction process. This request is subsequently handled by the host, while the warp awaits the return of the allocated address. During this interim, the Warp Scheduler registers memory allocation metadata for each thread, effectively overlapping the registration overhead. Once the allocated addresses are received from the host, the Warp Scheduler launches intra-warp rearrangement, transposing the memory layout within the warp memory chunks. Additionally, the Warp Scheduler registers each thread's offset into the Address Translator, facilitating subsequent memory access. Finally, the Warp Scheduler distributes the allocation addresses to each thread.

At the coarse level, rather than allocating a new address segment for warp allocation, the host identifies suitable free memory blocks within a pre-allocated super UVM address chunk and returns the starting address to the device. To fulfill the warp memory requests in the inter-warp rearrangement phase, which are organized into VABlocks of 2 MB each, the memory management system segments the continuous UVM address space into discrete units of the same size. To adeptly manage these segments, an OR tree-based management system is introduced. Once a suitable address is found, the host rearranges the inter-warp memory based on the layout of memory among warps.

B. OR Tree-Based Management

OR Tree-based management system is structured as a binary full tree, where each node represents the occupancy status of a UVM segment with a single bit, as depicted in Fig. 9. The OR tree-based structure streamlines operations and maintains a time complexity of $O(\log n)$, facilitating the rapid identification of free memory blocks without consuming extensive memory resources.

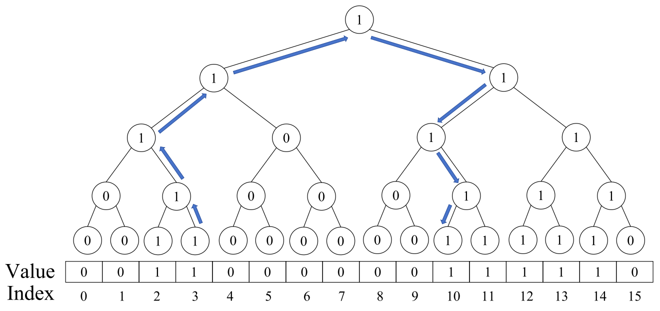


Fig. 9. OR Tree-based Management.

In the OR tree-based structure, the leaf nodes correspond to the states of UVM segments, where ‘1’ denotes idle and ‘0’ indicates occupied. Each non-leaf node in the tree represents the logical OR of the state of its two child nodes, indicates whether there is a free memory segment available within the memory range covered by its child leaf nodes. Updates to the nodes are carried out recursively upward, applying the logical OR operation until the root node is reached. The root has a global view, and in the top-down manner, the scope progressively narrows until a free segment is reached. This method ensures that each level of the tree reflects the overall availability of free memory across various scopes quickly.

To minimize memory fragmentation, OR tree-based management prioritizes locating a free segment with the minimum sequence number via the OR tree, ensuring the continuity of memory space. Specifically, the search for a free segment begins at the root node of the OR tree. The system consistently selects the leftmost non-zero child node at each level, descending down to the leaf node. The segment corresponding to this leaf node is identified as the free segment with the current minimum sequence number. Once the segment is allocated, the node states are updated recursively, starting from the leaf node and progressing upward to the root node. Both search and update operations maintain a computational efficiency with a time complexity of $O(\log n)$.

When the requirement extends to locate k consecutive free segments, the procedure first identifies the segment with the minimum sequence number to check if it can satisfy the need for k consecutive segments. If the initial segment does not suffice, the search proceeds from the last consecutive free segment to find its successor. This involves starting from the leaf node of the last free segment, traversing up the tree until a node is found that is the left child of its parent node and whose right sibling node indicates availability. The search then continues for the next free segment with the smallest sequence number starting from that sibling node. Fig. 9 depicts the search route for identifying the successor $node_3$. This method with the successor search continues until k consecutive free segments are located, with a worst-case time complexity of $O(k \log n)$.

C. Discussion

We provide a comprehensive comparison between AlignMalloc and existing approaches, focusing on both the memory management architecture and the underlying data structures.

Co-Management vs. GPU-Based Management: AlignMalloc’s host-device co-management overcomes the scalability limitations of previous GPU-based memory management systems, by offloading the branching demands of management tasks to the host. In this framework, the GPU collects memory requests from threads within a warp and forwards them to the host, leveraging the CPU’s serial processing capabilities. This offloading alleviates the computational burden on the GPU, allowing it to focus on parallel processing. Additionally, co-management addresses memory shortages by dynamically swapping inactive pages to the host when memory exceeds physical capacity. Finally, the use of dynamic data structures on the host side enhances scalability and robustness for large-scale memory allocations by replacing static pre-definition. In contrast, pre-defining memory slabs and management structures for large allocations in GPU-based systems presents significant challenges, increasing the likelihood of underestimating available resources. Consequently, this often leads to allocation failures.

OR Tree Vs. vEB Tree: While both the OR tree and vEB tree exhibit the same time complexity, the OR tree outperforms the vEB tree in terms of both operational efficiency and memory utilization in practical memory management tasks. The OR tree begins the search at the root node and continuously selects the leftmost non-zero child node at each level, descending toward the leaf nodes. Similarly, updates occur recursively from the leaf to the root. As a balanced binary tree, both the search and update operations involve traversing a logarithmic number of levels, with a time complexity of $O(\log\ n)$. In contrast, the vEB tree’s practical performance in Gallatin is hindered by the limited atomic capacity of GPUs, reducing its search complexity to $O(\log\ n)$ rather than the theoretical $O(\log \log\ n)$. Specifically, Gallatin eliminates the *maximum* and *minimum* values of tree nodes, necessitating more complex insertions for summaries, where the complexity can be expressed as the following recurrence: $T(n) = 2T(\sqrt{n}) + O(1)$. According to the Master’s Theorem, $T(n) = O(log\ n)$.

In terms of memory operations, the OR tree only requires bitwise *OR* operations on each node, whereas the vEB tree involves comparing the elements within the clusters and managing extreme values. This additional complexity in Gallatin leads to higher read/write frequencies and greater operational overhead. Moreover, the OR tree is highly efficient to construct, requiring only the allocation of a *boolean* array with all elements set to ‘1’, which takes $O(1)$ time. In contrast, constructing the vEB tree requires $O(n)$ time, with the overhead typically around 20 ms. Additionally, the OR tree is more memory-efficient, requiring only a *boolean* array of size n, while the vEB tree requires more complex memory structures, including extreme values and summary arrays.

VII. Experiments

A. Experimental Setup

The Setting of Comprehensive Evaluation: We provide a comprehensive evaluation of AlignMalloc, highlighting its performance in dynamic memory allocation and memory access through both synthetic benchmarks and real-world applications.

The evaluation compares AlignMalloc against SOTA allocators [12], [14], [16], focusing on efficiency, stability, and scalability in diverse scenarios.

AlignMalloc's allocation performance is first analyzed under singular allocation conditions, where it demonstrates strong capabilities in handling varying allocation sizes. These experiments are extended to scenarios involving oversubscription, where memory demands exceed available resources, to evaluate its stability under high pressure, as discussed in Section VII-B. To verify the stability of AlignMalloc's memory management under substantial memory demands, we employ random allocation tests in Section VII-E. Furthermore, by increasing the number of threads, we examine the advantages of our memory management system under extensive parallel threading conditions in Section VII-D.

Secondly, AlignMalloc's UVM access performance is evaluated against SOTA UVM policies. This is rigorously tested by enabling randomized memory access across substantial threads concurrently in Section VII-E, showcasing the effectiveness of AlignMalloc's warp-aware arrangement within the UVM. Detailed experimental comparisons and performance analysis under oversubscription further affirm our method's effectiveness in Sections VII-F and VII-G.

To assess its practical applicability, AlignMalloc is tested on the dynamic graph random walk algorithm, a widely used benchmark in graph clustering tasks involving large-scale datasets [25]. This application, commonly used for GPU dynamic allocator evaluation [16], provides a direct comparison with SyncMalloc and highlights AlignMalloc's superior performance in real-world dynamic allocation scenarios (Section VII-H). Furthermore, eight representative UVM memory access patterns [38] are analyzed to evaluate the effectiveness of AlignMalloc's warp-aware design in optimizing memory access for diverse real-world use cases, further validating the effectiveness of our warp-aware arrangement in UVM access in Section VII-I.

Baseline: For the comparison of memory allocation performance, several well-regarded allocators were utilized as baselines, including SyncMalloc [12], Gallatin [16], ScatterAlloc [33], and Ouroboros [14]. Traditional dynamic allocators such as XMalloc [32], Halloc [28], and FDGAlloc [34] have been reviewed in the dynamic allocation survey [1], which are not considered SOTA according to their findings. Additionally, some allocators, like Reg-eff [42], are incompatible with Nvidia's latest architectures. Consequently, these dynamic allocators are not included in our baselines.

In terms of evaluating UVM access performance, several established UVM prefetching policies were incorporated into the comparison, including SyncMalloc [12], ThrottlingFetch [39], DynamicACT [36], and Early-Adaptor [37]. ThrottlingFetch and DynamicACT involved hardware modifications, making them infeasible in actual GPU environments. Therefore, they were reimplemented at the application level for analysis. Specifically, ThrottlingFetch's GPU SM throttling was simulated on the host side by adjusting the maximum GPU power limit to control parallelism. Similarly, DynamicACT's access counter threshold adjustments were modeled by dynamically tuning

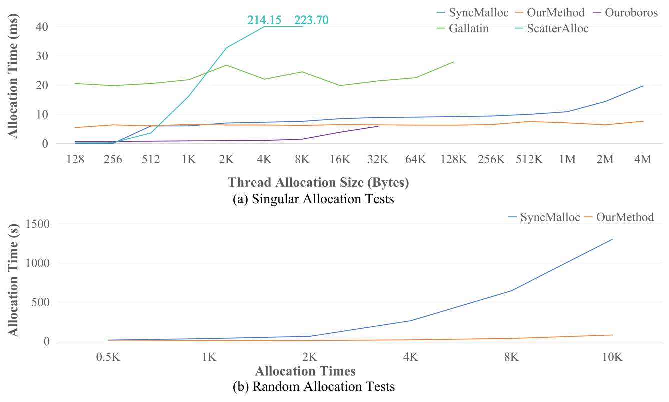


Fig. 10. Performance for Allocation Tests.

page placement thresholds in the host-side UVM kernel, based on page occupancy.

System specification: The evaluations were performed using an NVIDIA RTX 4090 GPU with 24 GB of DRAM, coupled with a high-performance CPU, the $14900KF$, featuring 24 cores. The hardware was operated within Ubuntu 20.04, using CUDA version 12.04. To obtain reliable and accurate performance results, each experimental setting was run 11 times. The initial run was discarded to mitigate the impacts of GPU warm-up, and the average time from the subsequent ten runs was then computed as the final results. Additionally, to evaluate the access efficiency of AlignMalloc, detailed memory tracking was conducted using *NVIDIA Nsight Systems*.

B. Singular Allocation Tests

We assess the performance of singular allocations where each thread's allocation size varies randomly between 2^n and 2^{n+1} bytes. Fig. 10(a) illustrates the allocation performance across 16 K threads, with the byte size denoted by 2^n on the X-axis. From the results, our method, AlignMalloc, excels in handling large memory allocations, significantly outperforming other methods in this area.

AlignMalloc can support allocations up to 4 MB per thread, which totals 64 GB, 2.5 times the GPU's physical memory. This is achieved through the host's co-management of dynamic memory and integration with the UVM system. In comparison, Ouroboros and Gallatin exhibit certain practical constraints. Ouroboros, which relies on pre-allocated memory slabs for different sizes, faces allocation failures under high memory demand as pre-allocated resources are exhausted. Although Gallatin theoretically supports up to 4 terabytes of memory using a 3-level vEB tree, practical limitations arise from the rapid depletion of data structures such as *memoryTable* and external fragmentation during large allocations. Gallatin mitigates internal fragmentation by splitting 16 MB segments into smaller blocks of various sizes to fit into memory requests. However, this approach leads to scattered available spaces, complicating the allocation of larger contiguous segments. In contrast, AlignMalloc addresses this by employing a host-side data structure with a dynamic linked-list approach instead of static pre-allocation and using a fixed 2 MB memory block per node to reduce the external fragmentation. Furthermore, AlignMalloc, integrated with UVM page placements, ensures efficient memory usage by evicting inactive pages to the host when memory exceeds physical capacity.

Within the allocation range of 2 KB to 4 MB, our method surpasses SyncMalloc, achieving a maximum speedup of $2.7\times$. As the allocation size increases, our method's advantage becomes more pronounced. This enhanced performance primarily stems from the efficiency of our OR tree-based management, which enables the quick location of free memory blocks with a time complexity of $O(\log n)$. Consequently, our method maintains a stable performance trend as allocated memory increases, proving particularly effective for large-scale memory demands in GPU.

When the allocation size is reduced to 2 KB or less, AlignMalloc's performance gradually matches or even slightly lags behind that of SyncMalloc. This is because SyncMalloc's sequential search approach that implements a first-fit strategy to locate available memory pages with a bitmap. Within this allocation range, the number of pages involved is relatively limited, enabling SyncMalloc to identify suitable pages more quickly than our approach, which requires $O(\log n)$ lookups for each operation.

AlignMalloc shows limitations in small allocation, particularly those below 512 B, where systems such as SyncMalloc, ScatterAlloc, and Ouroboros perform better. Despite their performance advantages, all lack the flexibility required for dynamic memory allocation in general-purpose applications. Essentially, these methods operate by estimating and pre-allocating fixed-size blocks of memory resources at initialization, which is not different from using the static pre-allocation function, *cudaMalloc*. Once the need for larger and more complex allocations arises, these pre-allocation decisions become increasingly difficult, leading to substantial overhead and memory fragmentation due to potential misestimation. This inherent limitation is highlighted by ScatterAlloc, which experiences fragmentation with memory requests above 256 B due to scattering allocations across predefined memory regions. Ouroboros, on the other hand, is also constrained by a maximum allocation limit of 0.5 GB due to the exhaustion of memory management resources. Similarly, SyncMalloc uses its slab-based method primarily to enhance efficiency for small allocations, but discontinues it once the allocation size surpasses 512 B due to inadequacy in accommodating more variable memory demands.

C. Random Allocation Tests

To simulate a more realistic scenario featuring multiple and randomly-sized memory operations, we implement stringent modifications to the random allocation benchmark originally proposed by Zhang et al. [12]. In this enhanced benchmark, 16 K threads execute a series of random allocations and deallocations multiple times. Diverging from the original benchmark where each thread consistently allocates the same block size, our modification assigns each thread a random allocation size ranging from 1 KB to 256 KB. Additionally, to better mimic environments with high memory occupancy rates, we increase the maximum number of memory blocks that can be allocated by the system from 256 to 200 K.

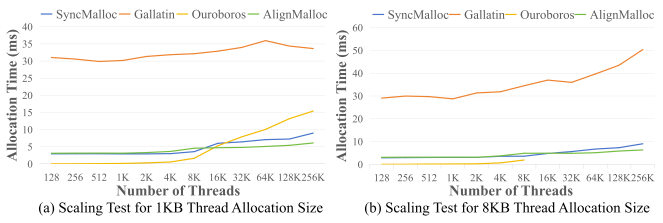


Fig. 11. Performance for Scaling Allocation Tests.

Due to their limitations in handling such intensive and varied memory management tasks, both Ouroboros and Gallatin quickly exhaust their memory management resources—after about 5 and 10 iterations, respectively. Consequently, Fig. 10(b) presents a comparative performance solely between our method and SyncMalloc.

In the random allocation benchmark, our method significantly outperforms SyncMalloc. Moreover, as the number of iterations increases, the superiority of AlignMalloc becomes increasingly evident, with achieving improvements of up to $300\times$. This remarkable efficiency of AlignMalloc is largely attributable to the stability and effectiveness of the OR-tree memory management, which excels in rapidly identifying free memory blocks. This capability ensures that the average overhead for each iteration remains stable at about 7 ms.

D. Scaling Allocation Tests

In scaling experiments, we evaluate the performance of AlignMalloc across a spectrum of parallelism, expanding the number of threads from 128 to 256 K, while maintaining a consistent thread allocation size of 1 K and 8 K bytes.

The results, depicted in Fig. 11, confirm that AlignMalloc consistently maintains robust performance as computing task scales up, showing no significant degradation even at higher scales. Remarkably, in large-scale parallel tasks involving 32 K to 256 K threads, AlignMalloc outperforms all other SOTA methods. For smaller-scale tasks, specifically when thread numbers range from 1 K to 4 K, Ouroboros shows superior performance due to its statically pre-allocated memory blocks, which enables rapid responses to allocation requests. However, as the thread number increases, Ouroboros encounters difficulties in adapting its pre-allocated blocks to meet the increasing requests and subsequently faces depletion of these resources, which leads to frequent rescheduling within its hierarchical structure. At a thread allocation size of 8 KB, these challenges within the memory management structures have even led to program crashes, as shown in Fig. 11(b).

At lower scales of tasks, the performance of AlignMalloc and SyncMalloc is comparably effective, with AlignMalloc occasionally underperforming SyncMalloc initially. This is attributed to the finer granularity of AlignMalloc's warp-level memory management, which, despite enhancing intra-warp memory

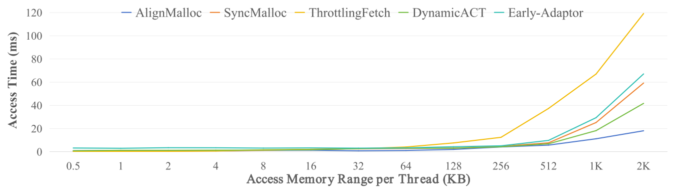


Fig. 12. Memory Access Performance with Existing UVM Policies.

management performance with SM's underlying registers, incurs greater overhead on the host due to managing the increased number of warps. However, as the thread scale increases, the efficiency of AlignMalloc's OR Tree-based system, which optimizes the locating free memory to logarithmic time complexity, gradually surpasses SyncMalloc.

E. Random Access Tests

In the random access tests, 1 K threads concurrently access the dynamically allocated memory, with each thread making a total of 100 random accesses. The memory access range for each thread varies from 0.5 KB to 2 MB, comparing AlignMalloc's performance against several SOTA UVM policies, as shown in Fig. 12.

The results indicate that while AlignMalloc does not exhibit a significant advantage in small-scale memory accesses, it progressively outperforms all other UVM policies as the memory access scale increases, achieving performance improvements of up to 6.6×. Specifically, for small memory accesses ranging from 0.5 KB to 4 KB, AlignMalloc performs slightly less effectively compared to SyncMalloc and ThrottlingFetch but outperforms DynamicACT and Early-Adaptor. This limitation arises because the memory access scale does not reach the UVM scheduling threshold required to activate AlignMalloc's warp-aware rearrangement, resulting in a memory access pattern similar to SyncMalloc. Furthermore, AlignMalloc incurs additional overhead from address translation during access. Similarly, DynamicACT experiences performance degradation at minor memory accesses due to its dynamic access counter threshold, which leads to the prefetching of irrelevant memory blocks when memory usage is low. Early-Adaptor shows the poorest performance due to significant overhead from monitoring page fault history and analyzing the page fault rate for each VABlock across multiple threads. In contrast, ThrottlingFetch performs best in this memory range by enhancing transfer performance through memory block compression. Although AlignMalloc does not outperform ThrottlingFetch in small memory accesses, the performance difference is minimal, with a gap of no more than 0.5 ms.

As the memory access scale increases, particularly beyond 16 KB, the benefits of AlignMalloc's memory rearrangement become evident. The system maintains stable overall time costs and achieves up to 2.3× better performance than the second-best prefetching policy, DynamicACT. This improvement is primarily driven by AlignMalloc's ability to reduce irrelevant prefetching memory blocks precisely. In contrast, DynamicACT dynamically adjusts page placement frequency under high memory usage, but fails to address the root cause of low prefetching hit rates, limiting its improvements. Similarly, Early-Adaptor's method of monitoring page fault rates fails to improve the prefetching hit rate effectively. ThrottlingFetch reduces thrashing by throttling the SM, but this only delays prefetching tasks. Additionally, the reduction in concurrency causes a significant decline in overall performance.

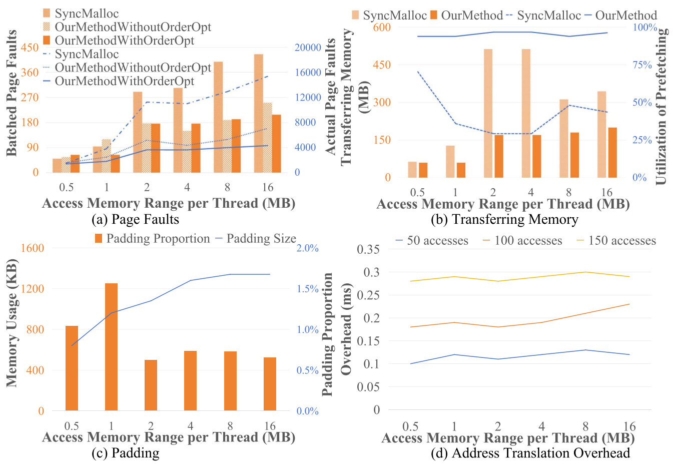


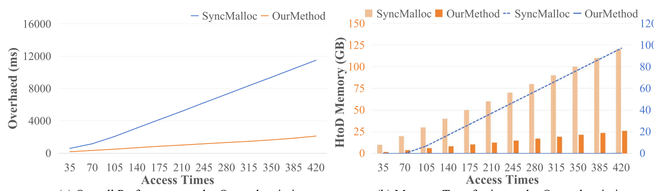
Fig. 13. Performance for Memory Access Tests with Large Scales.

F. Analysis for AlignMalloc's Large-Scale Access

To comprehensively evaluate the performance details for large-scale access, we expand the memory access range to 16 MB. The performance metrics collected by *Nsight Systems* include actual/batched page faults, memory transferred, and the utilization of prefetching pages. Additionally, we assess the overhead associated with padding in the inter-warp rearrangement and the effectiveness of pattern order optimization in the intra-warp rearrangement for AlignMalloc. For ease of evaluation, the number of threads is set at 128.

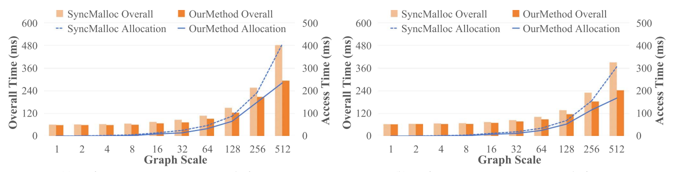
Fig. 13(a) shows a significant reduction in the number of page faults with AlignMalloc compared to SyncMalloc. As the memory access scale increases, the reduction becomes increasingly pronounced, reaching up to 2/3 that of SyncMalloc, which indicates that AlignMalloc can enhance the prefetching efficiency for large-scale accesses. Furthermore, the number of page faults with AlignMalloc stabilizes once the memory scale exceeds 2 MB. This stability is attributed to the optimal alignment granularity achieved by both intra and inter-warp rearrangement, which effectively syncs with UVM scheduling and maintains the independence of access among warps. Consequently, AlignMalloc shows promising robustness in large-scale memory operations.

Memory pattern order optimization shows a potential reduction in actual page faults from 10% to 30%. However, this optimization had a minimal impact on batched page faults. The size of batched faults typically depends on the hardware and the build environment, but is additionally adjusted dynamically according to runtime logic. Despite the reduction in actual page



(a) Overall Performacne under Oversubscription (b) Memory Transferring under Oversubscription

Fig. 14. Performance under Oversubscription Scenarios.



(a) Performance on NVIDIA 4090 Platform (b) Performance on NVIDIA A800 Platform

Fig. 15. Performance in Graph Clustering across Different Platforms.

faults, the modest decrease does not significantly influence the overall number of batched page faults.

Fig. 13(b) demonstrates that AlignMalloc significantly reduces the amount of UVM memory transfers by approximately 50% compared to SyncMalloc. Furthermore, it achieves nearly 100% actual utilization of prefetching pages, a substantial increase from about the 40% efficiency in SyncMalloc. The results show our AlignMalloc's ability to effectively suppress irrelevant page prefetching and enhance the hit rate of prefetching, thereby boosting prefetching efficiency overall.

Additionally, Fig. 13(c) reveals that the overhead associated with the padding of inter-warp rearrangement in AlignMalloc is minimal and acceptable, occupying about 1 MB of GPU physical memory space, which represents roughly 1% of the memory utilized.

Fig. 13(d) illustrates the address translation overhead introduced by AlignMalloc as threads access large memory ranges, with 100, 150, and 200 access iterations. The results demonstrate that the overhead remains minimal and relatively stable as the memory range increases. Specifically, for every 100 accesses, the address translation overhead for 0.1K threads stays at around 0.2 ms, much smaller than the GPU's page fault time. Additionally, as indicated in Fig. 13(a), AlignMalloc reduces the number of page faults significantly as the memory range accessed increases, while the translation overhead remains stable and negligible. Thus, AlignMalloc's performance advantages overshadow the translation overhead.

G. Memory Access Under Oversubscription

To assess performance under oversubscription, we expand the memory access range to 120 MB per thread. Each thread performs 420 random accesses within this extensive interval, a setup specifically designed to assess both the overhead and the volume of memory transferred in UVM.

AlignMalloc significantly enhances performance, achieving a 5.5× improvement compared to SyncMalloc, depicted in Fig. 14(a). Furthermore, the memory imported from the host is reduced by 4.5× in AlignMalloc in Fig. 14(b). When access times reach 100, SyncMalloc rapidly depletes GPU memory resources, which subsequently triggers eviction and incurs substantial overhead. In contrast, by benefiting from accurate prefetching, AlignMalloc postpones eviction until the very end of the access sequence. Consequently, AlignMalloc effectively delays the depletion of GPU resources, thereby diminishing the frequency of GPU-host memory thrashing under oversubscription.

H. A Graph Clustering Case

In a graph clustering scenario using varied dataset scales [25], AlignMalloc and SyncMalloc are evaluated based on overall overhead and UVM access time. The experiment is also conducted on the NVIDIA $A800$ platform to demonstrate AlignMalloc's adaptability, with results depicted in Fig. 15. At smaller graph scales, AlignMalloc performs comparably to SyncMalloc, showing a slight advantage since these scales do not fully exploit AlignMalloc's memory rearrangement benefits. However, as graph sizes increase, AlignMalloc begins to outperform SyncMalloc significantly, achieving up to a 2.6× improvement in UVM access performance and a 1.6× increase in overall performance. AlignMalloc's performance advantage is consistent across different platforms. However, at smaller scales, this advantage is less pronounced on the $A800$, potentially due to its superior memory bandwidth compensating for UVM limitations in memory access.

I. Access Performance in Real-World Cases

To validate the memory access performance of AlignMalloc within UVM, we conduct experiments using a diverse set of applications. These include six basic applications: General Matrix Multiply (GEM), stencil (STN), k-means (KMN), Breadth-First Search (BFS), Sparse Matrix-Vector Multiplication (SPV), and B+ tree (B+T), each representing different UVM memory access patterns such as Streaming, Thrashing, Part Repetitive, Most Repetitive, Repetitive-Thrashing, and Region Moving, as identified in [38]. Additionally, we incorporate two AI models, AlexNet [43] and Graph Convolutional Network (GCN) [44]. AlexNet primarily involves sequential and regular memory accesses with convolutional layers, while GCN exhibits sparse and random memory accesses, particularly during the computation of adjacency. This distinction in access patterns allows for a comprehensive evaluation across a wide range of memory access patterns and scenarios. We test AlignMalloc across various memory demands—1 GB, 4 GB, 20 GB, 32 GB, and 64 GB—with the 32 GB and 64 GB configurations exceeding the GPU's memory capacity by 33% and 167%, respectively. Throughout these tests, we consistently utilize 16 K concurrent threads.

The experimental results in Fig. 16 demonstrate substantial performance improvements in environments where memory is oversubscribed, with speedups ranging from 1.5× to 3.0×. At data scales below 20 GB, enhancements are observed across all applications, with the exception of GEM and STN. Both GEM and STN exhibit streaming access patterns typical of

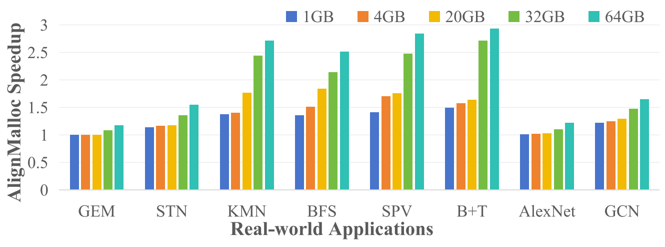


Fig. 16. Access Performance in Real-World Cases.

UVM memory usage. Specifically, GEM processes all virtual pages once, so that even if prefetched data is not immediately needed, it is ultimately utilized, thus eliminating any inefficiencies typically associated with irrelevant prefetching. Similarly, in the case of STN, while AlignMalloc effectively prefetches the necessary multidimensional grid points for each thread, the gains are relatively modest. This is because most resources prefetching by standard CUDA strategy are eventually used in later iterations. Nevertheless, both applications benefit from significant performance improvements under oversubscription scenarios, mainly due to reduced memory thrashing. As GPU memory resources near exhaustion, the LRU strategy leads to the eviction of unreferenced prefetched resources back to the host, causing program thrashing. In contrast, in AlignMalloc, prefetched resources are accessed shortly after their retrieval, substantially reducing thrashing and thereby enhancing the system's overall efficiency.

The B+T and SPV applications demonstrate the most significant enhancements with our method. The B+T application employs a 'region moving access pattern,' characterized by virtual memory chunks that are segmented into substantial address regions, with memory blocks within each region being referenced at varying frequencies. Similarly, SPV operates under a 'repetitive random sparsity pattern,' which introduces only selected parts of memory to the device. Our approach capitalizes on the SIMT architecture to accurately prefetch the memory needed by parallel threads within a warp, resulting in notable performance improvements in these scenarios. In the case of KMN, which exhibits a 'part repetitive access pattern', the scenario involves a temporal sequence where parts of virtual memory chunks are intermittently accessed with a certain probability. This pattern also aligns well with our prefetching strategy. However, due to the computationally intensive nature of KMN, the actual benefits observed are less pronounced compared to B+T and SPV.

Both AI models, AlexNet and GCN, exhibit performance improvements with AlignMalloc, particularly under oversubscription scenarios, achieving enhancements of up to 1.2 and 1.6×, respectively. GCN displays strong performance in memory accesses ranging from 1 GB to 20 GB. This is attributed to the effective reduction of irrelevant graph feature fetching during adjacency computation in graph convolution layers. Conversely, AlexNet benefits less from AlignMalloc because the sequential memory accesses in convolution layers are already efficient. Thus, improvements are mainly observed in the fully connected layers. Despite these differences, both applications experience significant advantages under oversubscription scenarios, as AlignMalloc reduces memory thrashing by promptly accessing prefetched resources.

VIII. Conclusion

This paper proposes AlignMalloc, a novel dynamic management system that employs a warp-aware memory rearrangement strategy for large-scale applications to align with the UVM prefetching, marking the first comprehensive effort to address both allocation and access performance in the GPU-based dynamic allocation applications. To uncover the underlying causes of the access inefficiencies in UVM, we conduct comprehensive experimental analysis, identifying the critical misalignment between dynamic memory access and the static nature of the prefetching setup, which leads to decreased hit rates and increased prefetching pages. To address the fundamental discrepancy, AlignMalloc introduces a warp-aware memory rearrangement strategy at two granularity levels, coordinating the prefetching scheduling and its scopes. To seamlessly integrate this strategy into dynamic allocation management, an OR tree-based architecture within a host-co-managed framework is introduced, which allows for an operational time complexity of $O(\log n)$ in memory allocation processes at the warp level, without requiring excessive memory space. Our experimental results demonstrate that AlignMalloc significantly outperforms existing dynamic allocators in terms of both dynamic allocation and memory access performance in large-scale scenarios.

AlignMalloc operates at the application level and integrates seamlessly with the existing UVM system, minimizing interference with other applications. While the current design performs well across a wide range of workloads, its effectiveness for fine-grained memory access is limited by hardware-level constraints in the UVM prefetching mechanism. In future work, we plan to further enhance AlignMalloc's adaptability to small-scale memory accesses through closer integration with the UVM runtime.

Acknowledgment

The authors express their gratitude to the anonymous reviewers for their insightful and constructive comments, which have significantly improved the manuscript.

References

[1] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on GPUs slow? A survey and benchmarks," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 219–233.
[2] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "Snakebyte: A TLB design with adaptive and recursive page merging in GPUs," in *Proc. IEEE Int. Symp. High- Perform. Comput. Archit.*, 2023, pp. 1195–1207.
[3] Z. Pan, S. He, X. Li, X. Zhang, R. Wang, and G. Chen, "Efficient maximal biclique enumeration on GPUs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2023, pp. 1–13.
[4] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "faimGraph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 754–766.
[5] M. Pham, Y. Tu, and X. Lv, "Accelerating BWA-MEM read mapping on GPUs," in *Proc. 37th Int. Conf. Supercomputing*, 2023, pp. 155–166.

[6] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, “TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs,” in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 90–106.

[7] X. Chen and Arvind, “Efficient and scalable graph pattern mining on GPUs,” in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 857–877.

[8] Y. Xing, Y. Li, Z. Wang, Y. Xu, and J. C. Lui, “LightTraffic: On optimizing CPU-GPU data traffic for efficient large-scale random walks,” in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 882–895.

[9] Y. Zhu, R. Ma, B. Zheng, X. Ke, L. Chen, and Y. Gao, “GTS: GPU-based tree index for fast similarity search,” *Proc. ACM Manage. Data*, vol. 2, no. 3, pp. 1–27, 2024.

[10] D. Li et al., “A memory-efficient hybrid parallel framework for deep neural network training,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 577–591, Apr. 2024.

[11] P. Dalmia, R. Mahapatra, J. Intan, D. Negrut, and M. D. Sinclair, “Improving the scalability of GPU synchronization primitives,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 1, pp. 275–290, Jan. 2023.

[12] J. Zhang, F. Wu, H. Jiang, G. Cheng, G. Chen, and Q. Wang, “SyncMalloc: A synchronized host-device co-management system for gpu dynamic memory allocation across all scales,” in *Proc. 53rd Int. Conf. Parallel Process.*, 2024, pp. 179–188.

[13] J. Lee et al., “Occamy: Memory-efficient GPU compiler for DNN inference,” in *Proc. 60th ACM/IEEE Des. Automat. Conf.*, 2023, pp. 1–6.

[14] M. Winter, D. Mlakar, M. Parger, and M. Steinberger, “Ouroboros: Virtualized queues for dynamic memory management on GPUs,” in *Proc. ACM Int. Conf. Supercomputing*, 2020, pp. 1–12.

[15] M. Pham et al., “Dynamic memory management in massively parallel systems: A case on GPUs,” in *Proc. ACM Int. Conf. Supercomputing*, 2022, pp. 1–13.

[16] H. Mccoy and P. Pandey, “Gallatin: A general-purpose GPU memory manager,” in *Proc. ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, 2024, pp. 364–376.

[17] B. Mayr, A. Weinrauch, M. Parger, and M. Steinberger, “Are van EMDE boas trees viable on the GPU?,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2021, pp. 1–7.

[18] NVIDIA, “Cuda runtime API - v12.5.1,” Mar. 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[19] B. Gao, Q. Kang, H.-W. Tee, K. T. N. Chu, A. Sanaee, and D. Jevdjic, “Scalable and effective page-table and TLB management on NUMA systems,” in *Proc. USENIX Annu. Tech. Conf.*, 2024, pp. 445–461.

[20] R.-H. Li, J. X. Yu, L. Qin, R. Mao, and T. Jin, “On random walk based graph sampling,” in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 927–938.

[21] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, “Crum: Checkpoint-restart support for Cuda’s unified memory,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 302–313.

[22] T. Allen and R. Ge, “In-depth analyses of unified virtual memory system for GPU accelerated computing,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–15.

[23] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, “Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory,” in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 224–235.

[24] R. Rossi and N. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *Proc. AAAI Conf. Artif. Intell.*, 2015, p. 4292–4293.

[25] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, “Terrace: A hierarchical graph container for skewed dynamic graphs,” in *Proc. 2021 Int. Conf. Manage. Data*, 2021, pp. 1372–1385.

[26] T. Allen, B. Cooper, and R. Ge, “Fine-grain quantitative analysis of demand paging in unified virtual memory,” *ACM Trans. Archit. Code Optim.*, vol. 21, no. 1, pp. 1–24, 2024.

[27] V. Seshadri et al., “Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 1–22, 2015.

[28] A. V. Adinetz and D. Pleiter, “Halloc: A high-throughput dynamic memory allocator for GPGPU architectures,” in *Proc. GPU Technol. Conf.*, 2014, vol. 152.

[29] I. Gelado and M. Garland, “Throughput-oriented GPU memory allocation,” in *Proc. Princ. Pract. Parallel Program.*, 2019, pp. 27–37.

[30] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, “NBBS: A non-blocking buddy system for multi-core machines,” *IEEE Trans. Comput.*, vol. 71, no. 3, pp. 599–612, Mar. 2022.

[31] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications,” *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[32] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, “XMalloc: A scalable lock-free dynamic memory allocator for many-core machines,” in *Proc. 10th IEEE Int. Conf. Comput. Inf. Technol.*, 2010, pp. 1134–1139.

[33] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, “ScatterAlloc: Massively parallel dynamic memory allocation for the GPU,” in *Proc. Innov. Parallel Comput.*, 2012, pp. 1–10.

[34] S. Widmer, D. Wodniok, N. Weber, and M. Goesele, “Fast dynamic memory allocator for massively parallel architectures,” in *Proc. Workshop Gen. Purpose Processor Using Graph. Process. Units*, 2013, pp. 120–126.

[35] D. Ganguly, R. Melhem, and J. Yang, “An adaptive framework for oversubscription management in CPU-GPU unified memory,” in *Proc. Des., Automat. Test Europe Conf. Exhib.*, 2021, pp. 1212–1217.

[36] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, “Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 451–461.

[37] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon, and W. W. Ro, “Early-adaptor: An adaptive framework forproactive UVM memory management,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2023, pp. 248–258.

[38] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, “HPE: Hierarchical page eviction policy for unified memory in GPUs,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2461–2474, Oct. 2020.

[39] C. Li et al., “A framework for memory oversubscription management in graphics processing units,” in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 49–63.

[40] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, “Batch-aware unified memory management in GPUs for irregular workloads,” in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 1357–1370.

[41] NVIDIA, “Open-GPU-kernel-modules,” 2024. [Online]. Available: https://github.com/NVIDIA/open-gpu-kernel-modules/tree/main

[42] M. Vinkler and V. Havran, “Register efficient dynamic memory allocator for GPUs,” *Comput. Graph. Forum*, vol. 34, no. 8, pp. 143–154, 2015.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 84–90.

[44] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, “Graph convolutional networks: A comprehensive review,” *Comput. Social Netw.*, vol. 6, no. 1, pp. 1–23, 2019.

Jiajian Zhang received the master’s degree from the College of Computer Science and Technology, Zhejiang University, in 2020. He is currently working toward the PhD with the University of Liverpool, Liverpool, U.K. His research interests GPU architecture, parallel computing, and distributed systems.

Fangyu Wu (Member, IEEE) received the PhD degree in computer science and software engineering from the University of Liverpool, Liverpool, U.K., in 2020. She is currently an assistant professor with the Department of Intelligent Science, School of Advanced Technology, Xi’an Jiaotong-Liverpool University (XJTLU), Suzhou, China. From 2020 to 2022, she was a postdoctoral researcher with College of Computer Science and Technology, Zhejiang University and has joined XJTLU in 2023. Her research interests include computer systems for AI, parallel computing, and multi-modal Learning.

Hai Jiang (Member, IEEE) received the MA and PhD degrees in computer science from Wayne State University, Detroit, MI, USA, in 1995 and 2003, respectively. He is currently a professor with the College of Computer Science, Beijing University of Posts and Telecommunications. His research interests include high performance computing, system security, and quantum computing. He is also a member of ACM.

Qiufeng Wang (Member, IEEE) is currently a professor and also the head with the Department of Intelligent Science, School of Advanced Technology, Xi'an Jiaotong-Liverpool University (XJTLU), Suzhou, China. He received the PhD degree in pattern recognition and intelligence systems from the Institute of Automation, Chinese Academy of Sciences (CASIA), Beijing, China, in 2012. From 2012 to 2013, he was an assistant professor with the National Laboratory of Pattern Recognition (NLPR), CASIA. From 2013 to 2017, he was with Microsoft and has joined XJTLU in 2017. His research interests include systems for AI, pattern recognition, and high performance computing. He was the recipient of the Presidential Scholarship of Chinese Academy of Sciences.

Genlang Chen received the PhD degree in computer science from Zhejiang University, China, in 2012. From 2014 to 2015, he was a researcher of computer engineering with the University of Arkansas. He is currently a professor with NingboTech University, Ningbo, China. His research interests include data mining, machine learning, and Big Data application, which include deep learning and parallel computing, for healthcare, multiscale modeling of complex networks, and programming models on hybrid computer networks.

Guangliang Cheng received the PhD degree from the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences, Beijing, China. He is currently an associate professor with the Department of Computer Science, University of Liverpool, Liverpool, U.K. He was a postdoctoral researcher with the Institute of Remote Sensing and Digital Earth, Chinese Academy of Sciences, China. He was a vice research director with SenseTime. His research interests include high-performance computing, robotics, and autonomous driving.

Eng Gee Lim (Senior Member, IEEE) received the BEng (with Hons.) and PhD degrees in electrical and electronic engineering from Northumbria University, Newcastle, U.K., in 1998 and 2002, respectively. Since 2007, he has been with Xi'an Jiaotong-Liverpool University, Suzhou, China, where he was formally the head of EEE Department and the University dean of research and graduate studies. He is currently the Inaugural School dean of advanced technology, Inaugural director with AI University Research Centre and also a professor with the Department of Communications and Networking. He has authored or caouthored nearly 200 refereed international journals and conference papers. His research interests include computer architectures, smart-grid communication, wireless communication.

Keqin Li (Fellow, IEEE) received the BS degree in computer science from Tsinghua University, Beijing, China, in 1985, and the PhD degree in computer science from the University of Houston, Houston, TX, USA, in 1990. He is currently a SUNY distinguished professor with the State University of New York and national distinguished professor with Hunan University, China. He has authored or co-authored more than 1130 journal articles, book chapters, and refereed conference papers and holds nearly 80 patents announced or authorized by the Chinese National Intellectual Property Administration. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked 2) and career-long impact (ranked 4) based on a composite indicator of the Scopus citation database. From 2022 to 2024, he was listed in ScholarGPS Highly Ranked Scholars and is among the top 0.002% out of more than 30 million scholars worldwide based on a composite score of three ranking metrics for research productivity, impact, and quality in the recent five years. From 2023 to 2024, he was listed in Scilit Top Cited Scholars and is among the top 0.02% out of more than 20 million scholars worldwide based on top-cited publications. He was the recipient of the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023, IEEE Region 1 Technological Innovation Award (Academic) in 2023, the 2022-2023 International Science and Technology Cooperation Award and the 2023 Xiaoxiang Friendship Award of Hunan Province, China. He is also a member of the SUNY Distinguished Academy, AAAS fellow, IEEE fellow, AAIA fellow, ACIS fellow, AIIA fellow, member of the European Academy of Sciences and Arts, and member of Academia Europaea (Academician of the Academy of Europe).