Journal of Systems Architecture 162 (2025) 103389

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Beyond VABlock: Improving Transformer workloads through aggressive prefetching

Jane Rhee [a], Ikyoung Choi [a], Gunjae Koo [b], Yunho Oh [b,*], Myung Kuk Yoon [a,*]

[a] Ewha Womans University, Seoul, 03760, Republic of Korea
[b] Korea University, Seoul, 02841, Republic of Korea

ARTICLE INFO

Keywords:
Unified virtual memory
Memory oversubscription
Graphics processing units
Large language models
Demand paging
Prefetching
Real-time analysis

ABSTRACT

The memory capacity constraint of GPUs is a major challenge in running large deep learning workloads with their ever increasing memory requirements. To run a large Transformer model with limited GPU memory, programmers need to manually allocate and copy data between CPU and GPUs. This programming burden is eased by Unified Virtual Memory (UVM), which automatically manages data transfer through its demand paging scheme. However, using UVM can cause performance degradation, especially under memory oversubscription. In this paper, we analyze the memory behavior of inference in large Transformer models using real hardware and the open-source NVIDIA UVM driver. The default Tree-Based Neighborhood (TBN) prefetcher in the UVM driver supports page prefetching within a 2MB virtual address block (VABlock), but it only detects locality within a VABlock, limiting its effectiveness for large models. Our analysis reveals that this locality extends beyond the VABlock, which the default prefetcher cannot exploit. To address this, we propose a block-aware prefetcher that prefetches multiple contiguous VABlocks with greater aggressiveness. Our evaluation shows that this approach delivers an average 2.7x performance improvement over the default TBN prefetcher when GPU memory is oversubscribed.

1. Introduction

Over the past few years, the deep learning field has been experiencing a major upheaval fueled by the fast-paced growth of Transformer models [1–3]. Characterized by its distinct self-attention mechanism, a Transformer model is a neural network that learns the context of sequential data and generates new data out of it. It was first developed to solve any task that transforms an input sequence into an output sequence, and has achieved success across a wide range of Natural Language Processing (NLP) tasks [4].

Since deep learning involves a huge amount of matrix multiplications, GPUs are optimized for both training and inference by leveraging parallel processing. With ongoing advancements in neural networks, these models are growing larger [5]. Bigger models usually perform better, as they are capable of comprehending complex patterns in the data, resulting in more accurate outputs [6]. This trend of larger models requires increased computation for both training and inference. With GPUs increasingly used to scale deep learning workloads, their limited memory capacity has become a significant constraint for executing large models [7].

Many open-source deep learning workloads follow the classic Copy-Then-Execute (CTE) programming model, requiring programmers to manually allocate and copy data between CPU and GPUs. This approach not only increases programming difficulties, but also confines the memory footprint to the capacity of GPU memory. To address these challenges, Unified Virtual Memory (UVM) is introduced.

Implemented in NVIDIA [8] and AMD [9] GPUs, UVM provides a single virtual address space shared between CPU and GPU. It automatically transfers data between them using a demand paging scheme [10]. When the GPU accesses a physical page that is not resident in its memory, a page fault is triggered and the data is migrated to the GPU memory. Pages can also be evicted from the GPU when it runs out of storage capacity. This allows UVM to accommodate GPU applications with a memory footprint larger than the available GPU physical memory [11]. We find these features beneficial for executing large deep learning workloads.

Despite these advantages, UVM presents two major challenges. The first is that the faulted warp stalls its execution while the corresponding page fault is serviced. When a thread creates a global memory request that results in a page fault, the warp containing the thread waits for the

* Corresponding authors.
E-mail addresses: jrhee1122@ewhain.net (J. Rhee), gurum789@ewhain.net (I. Choi), gunjaekoo@korea.ac.kr (G. Koo), yunho_oh@korea.ac.kr (Y. Oh), myungkuk.yoon@ewha.ac.kr (M.K. Yoon).

https://doi.org/10.1016/j.sysarc.2025.103389
Received 29 October 2024; Received in revised form 7 February 2025; Accepted 23 February 2025
Available online 4 March 2025

page to be loaded to the GPU memory. This delay from stalled warps induces significant page fault handling latency [12]. The second challenge is the increased overhead caused by memory oversubscription. When the memory footprint exceeds the GPU memory capacity, pages are evicted from GPU memory to CPU memory to make room for the incoming pages. After pages are evicted, the requested pages are then migrated to the GPU memory. The frequent page migration and eviction under memory oversubscription leads to significant performance overhead [13].

To mitigate UVM-related performance degradation, NVIDIA features a Tree-Based Neighborhood (TBN) prefetcher that proactively migrates pages that are likely to be accessed soon to the GPU memory [14]. UVM splits a contiguous bound of memory allocation, referred to as a virtual address range, into logical 2 MB virtual address blocks (VABlocks). The TBN prefetcher migrates pages to the GPU memory when the amount of GPU-resident pages in a subregion of a VABlock surpasses the predefined prefetch threshold. This approach allows pages to be prefetched into GPU memory along with the faulted pages, thereby preventing potential page faults within the VABlock.

In this work, we examine the underlying potential of prefetching in deep learning workloads by analyzing the performance effects of the TBN prefetcher using different prefetch thresholds on open-source GPT models [15]. To the best of our knowledge, this is the first study to assess the performance impact of the TBN prefetcher with varied prefetch aggressiveness in Transformer models. Our observations indicate that aggressive prefetching yields the best GPU performance for large Transformer models experiencing memory oversubscription. Further analysis reveals that these performance gains derive from the memory access patterns of Transformer models.

There are three distinct memory access characteristics for GPT models, each in the context of (1) patterns of page access, (2) patterns of VABlock access, and (3) patterns of virtual address range access. Pages within the same VABlock tend to be accessed randomly; however, due to spatio-temporal locality, this randomness remains confined within the same VABlock for a certain period of time before pages in another VABlock are accessed. As neighboring pages within a faulted VABlock are likely to be accessed soon, aggressively prefetching these pages proves to be effective. The trend in accessed VABlocks show a linear increase, while the virtual address ranges including these VABlocks are accessed in either an increasing or decreasing order. The default TBN prefetcher operates within a single VABlock, which prevents it from fully taking advantage of the memory access patterns on a larger scale. Thus, our analysis suggests an opportunity for a more effective prefetch scheme by exploiting the contiguity within and between virtual address ranges.

Memory access patterns like these are prominent in the forward pass, which is the dominant workload during inference, and are also observed during training. Building on this observation, we propose a new prefetching strategy for Transformer models, called the block-aware prefetcher, which accounts for memory access patterns and leverages the contiguity within and across virtual address ranges. The block-aware prefetcher improves GPU performance by concurrently migrating multiple contiguous VABlocks to GPU memory. Since VABlocks are accessed sequentially, prefetching VABlocks adjacent to the faulted VABlock helps mitigate anticipated page faults in nearby regions. Unlike prior approaches that restrict the migration scope to a single 2 MB VABlock, our method is the first to extend the prefetch scope beyond this limit, offering a more comprehensive solution to memory management [10,16,17].

This paper makes the following contributions:

- This paper presents the analysis of how different prefetch thresholds of the TBN prefetcher impact performance in deep learning workloads. We demonstrate how aggressive prefetching is effective for Transformer models.

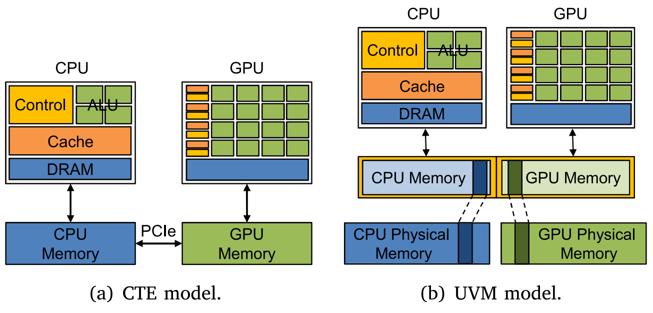


Fig. 1. Conventional CTE model and UVM model.

- This paper proposes the block-aware prefetcher, which prefetches multiple contiguous VABlocks to GPU memory and further minimizes potential page faults that may occur from them.
- The proposed prefetcher is evaluated on real hardware, demonstrating an average performance improvement of 2.7x over the baseline TBN prefetcher in cases of memory oversubscription.

The rest of this paper is organized as follows. Section 2 provides background of UVM and its functionalities. Section 3 presents our analysis of Transformer models using UVM and motivates the need for a new page prefetch policy. Section 4 proposes the block-aware prefetcher based on our analysis. Section 5 evaluates the block-aware prefetcher, compares its performance with other prefetching strategies, and discusses potential areas of future research. Related work on UVM is discussed in Section 6 and Section 7 concludes this paper.

2. Background

In this section, we start with a brief overview of UVM, followed by a detailed explanation of the page fault handling mechanism in UVM. Lastly, we describe the TBN prefetcher, a feature supported by the UVM driver to enhance memory management efficiency.

2.1. Unified virtual memory

In the CTE programming model for GPUs, programmers must explicitly allocate memory and manually copy data between CPU and GPU. While this approach is functional, it introduces significant programming complexities. To alleviate these challenges, NVIDIA [8] and AMD [9] introduced the concept of UVM. As shown in Fig. 1, UVM provides a single virtual address space that is shared between CPU and GPU [18]. With UVM, a single pointer is used to access both host and device memory, while the system automatically manages data by migrating it to the memory of the requesting processor. This automatic data migration eliminates the need for manual page migration, greatly reducing the programming burden.

In UVM, a demand paging scheme is implemented to support automatic data migration between CPU and GPU [10]. When a GPU attempts to access a physical memory page that is not resident in its memory, the GPU Memory Management Unit (GMMU) generates a page fault, prompting the migration of the corresponding page from CPU to GPU [19]. Demand paging loads pages only as needed, meaning that pages must either be referenced or prefetched to be migrated to GPU memory. Moreover, pages can be evicted to the host when GPU memory lacks enough space to store incoming pages. The detailed steps of page fault handling are explained shortly.

The demand paging technique in UVM provides programmers with two major advantages. Firstly, it eliminates the need for explicit data copying between CPU and GPU. Without demand paging, programmers would face the significant burden of manually managing data transfers between these processors. Secondly, demand paging enables UVM to

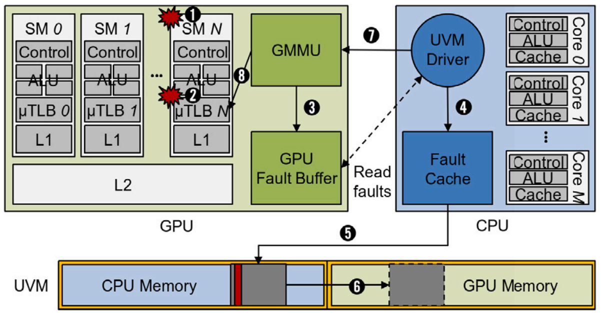


Fig. 2. Overview of page fault handling.

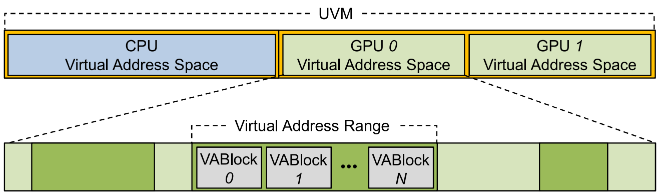

UVM
CPU Virtual Address Space
GPU 0 Virtual Address Space
GPU 1 Virtual Address Space
Virtual Address Range
VABlock 0
VABlock 1
VABlock N

Fig. 3. Memory hierarchy of UVM.

support memory oversubscription. In the CTE model, the memory footprint of a GPU application is limited to the physical memory capacity of the GPU, as programmers must manually ensure that the data size does not exceed the available device memory. If the required data size surpasses the available memory, the application will terminate with an out-of-memory (OOM) error. However, with demand paging, pages can be evicted from GPU memory to accommodate newly requested pages, allowing GPU applications to utilize a memory footprint larger than the available GPU physical memory [11,13].

2.2. Page fault handling

In demand paging, when threads scheduled on GPUs attempt to access physical memory pages that are not present in GPU memory, the GPU generates a page fault. This causes the GMMU to raise a page fault interrupt, which is then handled by the UVM driver. Fig. 2 shows the overall sequence of generating a page fault and how the UVM driver handles such a fault.

As shown in Fig. 2, when an active thread in the GPU generates a global memory request (1) and the page containing the requested data is not found in the corresponding μTLB (resulting in a μTLB miss) (2), the system then searches for the page in the page table. If the page is not found in the page table, indicating that it is not currently present in GPU memory, the GMMU raises a page fault interrupt. The page fault interrupt is handled in two phases, known as the top-half and the bottom-half Interrupt Service Routines (ISRs). The top-half ISR first responds to the interrupt and checks for actual pending page faults to make sure the bottom-half ISR for page fault handling is needed. If such faults are detected, the ISR disables further page faults to maintain the integrity of the GPU fault buffer. Meanwhile, all triggered page faults are written to the GPU fault buffer, and no new page faults can be generated until the buffer is managed (3). After page faults are disabled, the top-half ISR schedules a bottom-half ISR to manage the rest of the page fault handling.

Since page fault handling is very expensive in terms of latency, the UVM driver tends to process a group of page faults together rather than handling each one individually, a method known as *batch processing*. Thus, the page faults stored in the GPU fault buffer are fetched to the host-side fault cache as a group, referred to as a *fault batch* (4). This approach amortizes the overhead of multiple round-trip latencies over the PCIe and avoids invoking multiple ISRs in the operating system. The UVM driver retrieves a group of page faults until any of the three conditions are met: (1) when the number of retrieved page faults reaches a predefined batch size (*e.g.,* 256 faults), (2) when a predefined time limit is reached (*e.g.,* 30 ns), or (3) when all page faults in the GPU fault buffer have been fetched.

The page faults fetched in the fault cache can be scattered over a broad range of the corresponding virtual address space. The UVM driver groups these page faults into logical 2 MB blocks and services them in the granularity of a block, rather than individually servicing each 4 kB page (5). Handling page faults in the granularity of a single 4 kB page leads to frequent data migration between CPU and GPU, which increases the latency of the overall page fault handling process. Therefore, handling all page faults in the logical 2 MB block at a time can reduce the data migration overhead. After the page faults are grouped into corresponding 2 MB blocks, the pages are unmapped from the CPU page table, migrated to the device memory, and mapped to the GPU page tables (6). Once the pages are migrated, the UVM driver requests the GMMU to update the page tables and issue a fault replay (7). A fault replay clears out the waiting status of the μTLBs and resumes the memory requests of the prior miss. The UVM driver replays once per batch, causing all page faults in the fault batch replayed at the same time. The fault replay issued to the faulted μTLB resumes the memory request, and successfully finds the newly migrated pages (8).

2.3. Tree-based neighborhood prefetcher

While UVM offers benefits such as overcoming memory capacity limits and increased programmability, it also introduces notable delays when managing data between CPU and GPU. As GPUs operate using a Single Instruction Multiple Thread (SIMT) execution model, a group of threads, known as a *warp*, executes in lockstep [20]. Consequently, if any thread within a warp experiences a page fault, all other threads in the warp must stall until the page fault is handled. This page fault handling can take thousands of clock cycles, creating a major performance bottleneck [12]. The overhead becomes even more pronounced under memory oversubscription, which leads to frequent page migrations and evictions. To mitigate this overhead, the UVM driver includes a TBN prefetcher, which proactively migrates pages likely to be accessed soon into GPU memory.

In UVM, pages are managed using four conceptual abstraction layers: virtual address space, virtual address range, VABlock, and page. Fig. 3 shows these conceptual abstraction layers in the context of a configuration with two GPUs in the system. In general, a virtual address space represents the virtual memory of a single processor. In this case of UVM shared between CPU and two GPUs, each processor would have its own virtual address space, resulting in a total of three virtual address spaces. A virtual address space comprises multiple virtual address ranges, each corresponding to a contiguous bound of memory allocation. The UVM driver splits these memory allocations into logical 2 MB VABlocks, each consisting of 512 4 kB pages.

TBN prefetcher is designed to migrate multiple contiguous pages within the corresponding VABlock of the faulted pages. When servicing page faults, pages are first logically grouped into 2 MB VABlocks. Each VABlock is further divided into 64 kB virtual address regions, creating a full binary tree covering the 2 MB boundary. Fig. 4 shows the tree-based data structure of the TBN prefetcher. The root node of the tree represents a 2 MB VABlock, allowing the TBN prefetcher to manage memory space in 2 MB units. Its leaf nodes correspond to 64 kB regions, each consisting of 16 4 kB pages. With each leaf node covering a 64 kB region of a VABlock, the pages are prefetched at a granularity of 64 kB.

The operation of TBN prefetcher can be demonstrated with the example shown in Fig. 4. The first page fault in the leftmost leaf node N_0^0 prompts the UVM driver to identify the 64 kB region containing the

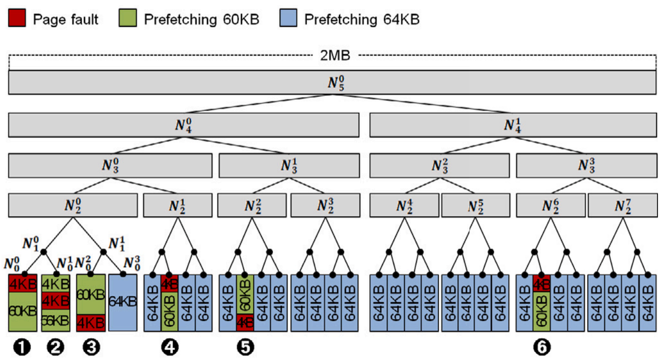


Fig. 4. Operation of TBN prefetcher.

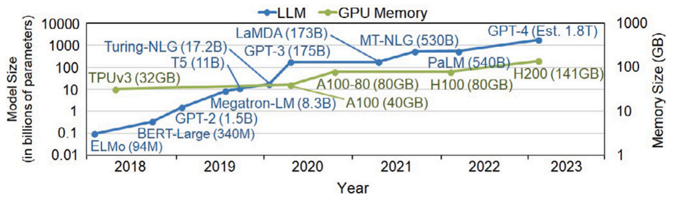


Fig. 5. Evolution of the increase in the model size of LLMs and GPU memory capacity.

Table 1
Transformer models executed with CTE programming and UVM.

GPU system	Programming model	Model size 355M	774M	1.5B	6.7B	13B
RTX 3090 [27]	CTE	○	○	×	×	×
	UVM	○	○	○	○	○
RTX A6000 [28]	CTE	○	○	○	×	×
	UVM	○	○	○	○	○

faulted page (❶). If any of the 16 pages in this region are migrated to GPU memory, the remaining pages are prefetched as well. Thus the remaining 60 kB of data in N_0^0 are prefetched along with the faulted page. The second (❷) and third (❸) page faults follow the same process. For each level of the tree (except for the leaf level), if the total size of GPU-resident pages exceeds the predefined threshold (*i.e.,* 51%) of the maximum memory capacity at that level, the remaining pages are prefetched to GPU memory. After migrating three regions of pages (N_0^0, N_0^1, and N_0^2) to GPU memory, the TBN prefetcher detects that more than the predefined threshold of N_2^0 is resident in the GPU memory. Therefore, N_0^3 is also prefetched to the GPU memory. The fourth page fault also triggers the corresponding 64 kB region to be prefetched (❹). The TBN prefetcher again notices that more than the predefined threshold of N_3^0 reside in the GPU memory, leading to the prefetching of all the remaining child nodes of N_2^1. The same process repeats for the fifth page fault, migrating the entire region covered by N_4^0 (❺). After the sixth page fault, more than the predefined threshold of the root node N_5^0 are migrated to the GPU memory, hence the remaining pages in the VABlock are prefetched to the GPU memory (❻).

3. Motivation and UVM analysis

In this section, we address the challenge of executing large Transformer models that rely on the conventional CTE programming model on GPUs. While UVM helps mitigate this limitation, the default TBN prefetcher restricts the performance due to its static prefetch threshold. To understand the reason behind this suboptimal performance, we analyze the detailed memory access patterns of large Transformer models using UVM.

3.1. Challenges in running large language models on GPUs

As one of the leading deep learning workloads today, Transformer models are widely recognized as one of the most common architecture for large language models (LLMs), and have achieved significant success in the field of NLP. In the current deep learning paradigm, deeper and larger models tend to outperform smaller models [21–23]. Larger models, integrated with larger training datasets, enhance the ability to comprehend complex patterns while minimizing the risk of overfitting, ultimately resulting in higher model accuracy [6,24]. Consequently, this trend towards bigger models requires increased computation for both training and inference, whether through larger model sizes, bigger datasets, or more training steps.

While computational resources have improved with each new generation of GPUs [25,26], existing memory management schemes still struggle to efficiently utilize these GPUs [7]. Moreover, limited memory capacity has become a major bottleneck in executing large deep learning applications on GPUs. As shown in Fig. 5, the size of LLMs continues to grow rapidly, while GPU memory capacity increases at a much slower pace. Many open-source LLMs rely on the classic CTE programming model, which manually manages memory allocation, preventing the execution of large Transformer models on GPUs due to limited memory capacity. The limitations of the CTE programming model are further exacerbated by the current trend of vastly increasing model sizes.

In this paper, we evaluate an open-source GPT model with five different sizes: 355 million (M), 774M, 1.5 billion (B), 6.7B, and 13B parameters [15]. These models are tested on two GPU systems: an RTX 3090 [27] and an RTX A6000 [28]. The detailed evaluation methodology is described in Section 5.1. The models rely on the conventional CTE programming model, which limits their execution if the GPU lacks sufficient memory to load all the data. In such cases, the GPU encounters an OOM error and is unable to run the model.

Note that even under memory oversubscription, the CTE programming model, where programmers manually manage memory by evicting and allocating data using APIs such as `cudaMemAdvise()` and `cudaMemPrefetchAsync()`, may achieve higher throughput than UVM and help avoid OOM errors. However, this approach sacrifices programmability, flexibility, and portability, making software more difficult to maintain, scale, and deploy across different GPU architectures [29]. Additionally, reliance on these functions often ties the code to specific GPUs, reducing portability and requiring rewrites for different configurations. Moreover, based on our efforts, no available open-source GPT models are designed to automatically manage data using the CTE programming model under memory oversubscription. In other words, current available open-source GPT models encounter OOM errors when model parameters exceed the capacity of the given GPU memory.

Table 1 shows whether the five different sizes of model can be executed on the two GPU systems, respectively. Using the default CTE programming model, only the two smallest models are executed on an RTX 3090, while an RTX A6000 can handle one more model with 1.5B parameters, out of the five models. Such OOM errors can be overcome through UVM by utilizing the demand paging scheme that supports memory oversubscription. Demand paging enables automatic data migration between CPU and GPU, allowing pages to be migrated from device to host when memory is oversubscribed, thereby freeing up space for newly requested data. This allows applications using UVM to use more memory than what the GPU's physical memory can accommodate. As shown in Table 1, by leveraging UVM, models larger than the GPU memory capacity can be executed without encountering OOM issues. For instance, the 13B model can be run on a single GPU with UVM, whereas the model is known to originally require, though not officially stated, two GPUs, each equipped with 40 GB of device memory.

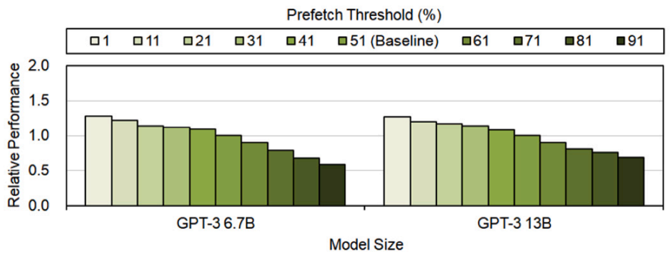


Fig. 6. Performance comparison of various prefetch thresholds for the TBN prefetcher in GPT-3 6.7B and 13B models. Performance is normalized to the baseline, which is the TBN prefetcher with a default 51% threshold for each model size. Configurations ranging from 1 to 91 represent the different prefetch thresholds used with the TBN prefetcher.

Despite supporting memory oversubscription, UVM comes with two major challenges. The first is that kernel execution stalls during fault-driven page migration. When a thread creates a global memory request (resulting in a page fault), the corresponding warp waits for the page to be retrieved to the GPU. The warp resumes processing only after the UVM driver issues a fault replay. Prior work indicates that this page fault latency can reach 30 to 45 μs, creating a significant performance bottleneck [10].

The second challenge is the increased overhead caused by memory oversubscription. When the GPU memory is fully allocated and new memory requests occur, UVM evicts pages from device to host in order to make room for the incoming pages. The evicted pages are swapped back to the host in the granularity of a 2 MB VABlock, which are selected according to the LRU policy [30]. After pages are evicted, the UVM driver then migrates the newly requested pages to GPU. These frequent page migration and eviction under memory oversubscription incur significant overhead in page fault handling.

To mitigate this overhead, NVIDIA GPUs utilize the default TBN prefetcher, which leverages the spatio-temporal locality between pages within a VABlock. In the following subsection, we provide an in-depth analysis of the TBN prefetcher's performance on deep learning workloads. To the best of our knowledge, this is the first work to explore the impact of the TBN prefetcher on such workloads.

3.2. Performance effects of TBN prefetcher

TBN prefetcher is designed to proactively migrate pages within a faulted VABlock, preventing further page faults from the same VABlock. As described in Section 2.3, TBN prefetcher operates with a predefined prefetch threshold. For levels higher than the leaf level, all pages covered at that level are migrated together if more than the predefined threshold is resident in the GPU memory. Whereas the prefetch threshold of the UVM driver is originally set to 51%, we discover that applying the static default prefetch threshold does not necessarily lead to the best performance. To find out the optimal prefetch threshold for Transformer models, we conduct experiments with modified prefetch thresholds, which consequently affects the prefetch aggressiveness. The performance of inference for the two largest GPT models listed in the previous subsection, with 6.7B and 13B parameters, is evaluated on an RTX 3090 system, both of which are heavily impacted by memory oversubscription.

Fig. 6 shows the relative performance of the two large models with various prefetch thresholds. The performance is normalized to the baseline prefetch threshold (*i.e.,* 51%). In both cases, aggressive prefetching (*i.e.,* lower prefetch threshold) results in higher performance, with an average increase of 24.3% compared to the baseline prefetch threshold. With the lowest prefetch threshold (*i.e.,* 1%), a single page fault in a 2 MB VABlock triggers migrating all the remaining pages (2044 kB) at once. In contrast, the highest prefetch threshold (*i.e.,* 91%) shows the lowest performance. In this case, only after loading 30 out of the total 32 leaf nodes into GPU memory does the remaining pages (128 kB) get prefetched, resulting in only a small amount of pages being prefetched.

Based on our evaluation results, aggressive prefetching yields the best performance for large Transformer models experiencing memory oversubscription, a scenario common in most deep learning workloads today [31]. Therefore, in the following subsection, we examine the memory access patterns of page faults to discover the reason behind this result.

3.3. Memory access patterns

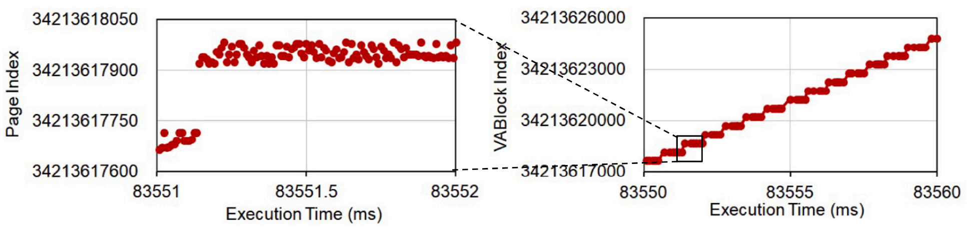
In the previous subsection, we discovered that performance improves for larger deep learning workloads as pages are more aggressively prefetched into GPU memory. For better understanding, we measure page faults generated during inference tasks in GPT models in the context of (1) patterns of accessed pages, (2) patterns of accessed VABlocks, and (3) patterns of accessed virtual address ranges. Given that memory access patterns are similar across different model sizes, we focus on the memory access patterns of the GPT-3 13B model.

Fig. 7(a) shows the trend of page faults when executing the GPT-3 13B model under memory oversubscription conditions. The left figure shows the page indices of individual page faults, while the right figure shows the page faults on a larger scale with VABlock indices. Though the pages are randomly touched in the order of occurrence, pages from the same VABlock tend to be faulted together due to the locality of neighboring pages. This indicates that pages in a VABlock are accessed and fetched together; thus, page faults are concentrated in one or more batches for a period of time. As neighboring pages within a faulted VABlock are likely to be accessed soon, aggressively prefetching these pages can effectively reduce the number of batches required for the corresponding VABlock. Also, interestingly, the trend in accessed VABlocks shows a linear increase, as illustrated in the right figure of Fig. 7(a). To comprehend this pattern on a larger scale, we then examine the trend of page faults among virtual address ranges.

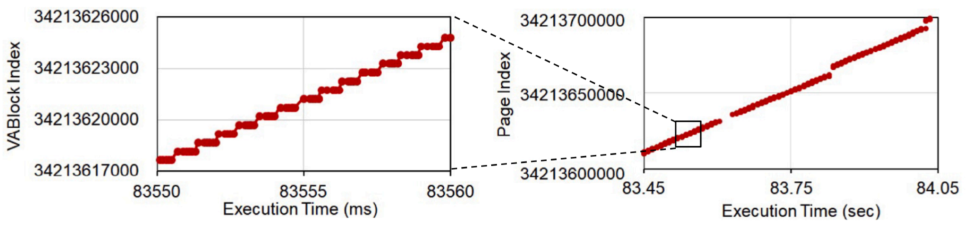
The distribution of page faults across virtual address ranges is illustrated in the right figure of Fig. 7(b) and throughout Fig. 7(c). We observe two major trends of page faults displayed in each figure. The right figure in Fig. 7(b) shows the increasing pattern of accessed virtual address ranges, while Fig. 7(c) shows the decreasing pattern. Notably, in Fig. 7(c), although virtual address ranges are accessed in a descending order, the VABlocks composing each virtual address range exhibit a pattern of linearly increasing access. This indicates that locality extends beyond the 2 MB VABlock limit, and that contiguity in memory access exists not only between pages, but also within and between virtual address ranges. However, the default TBN prefetcher operates within a single VABlock, which prevents it from fully leveraging the memory access patterns on a larger scale. Therefore, our analysis suggests an opportunity for a more effective prefetch scheme by exploiting the contiguity both within and between virtual address ranges.

The final key observation is that, while regularity is observed at the page and VABlock levels, virtual address ranges exhibit irregularities due to varying distances between them. These ranges represent contiguous memory allocations, uniquely distributed across the virtual address space with varying lengths initialized at the program's start. This underscores the benefits of UVM over manual page management, as using APIs such as `cudaMemAdvise()` and `cudaMemPrefetchAsync()` demands programmers to handle this complex memory distribution, requiring significant expertise in memory layout. UVM, on the other hand, offers a more efficient and flexible approach to overcoming these challenges.

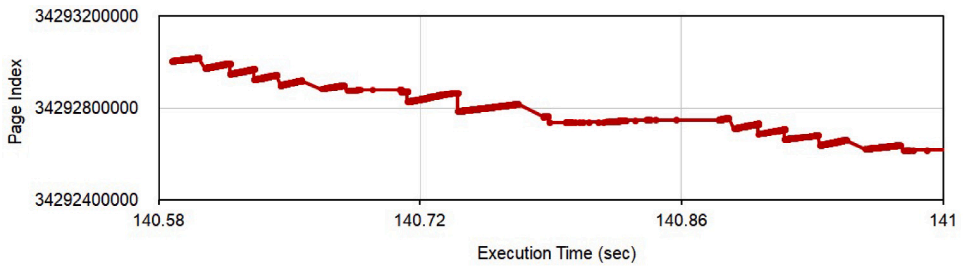
In summary, our analyses of the memory access patterns of Transformer models provide the following insights: (1) pages within the same VABlock are accessed together until pages from another VABlock are accessed, (2) VABlocks are accessed sequentially, and (3) virtual address ranges are accessed in either an ascending or descending order, showing locality among these ranges. These observations lead us to propose a block-aware prefetching scheme that leverages the locality extending beyond a single VABlock.



(a) Page faults in page level (left) and page faults in VABlock level (right).



(b) Page faults in VABlock level (left) and increasing pattern of accessed virtual address ranges (right).



(c) Decreasing pattern of accessed virtual address ranges.

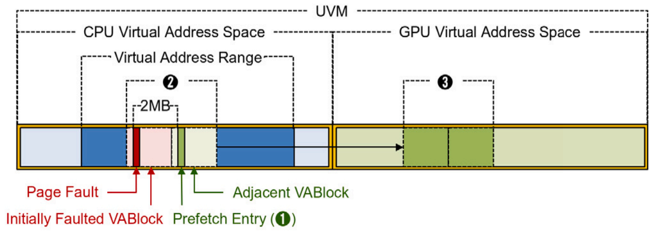
Fig. 7. Memory access patterns of pages, VABlocks, and virtual address ranges during the inference of GPT-3 13B model.

4. Block-aware prefetcher

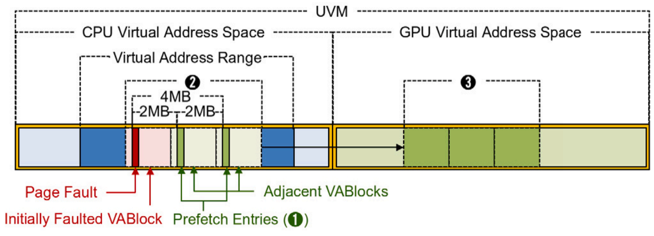
Based on our analysis, we propose a block-aware prefetcher that simultaneously migrates multiple contiguous VABlocks within the same virtual address range. This prefetching scheme is designed with an understanding of the memory access patterns of Transformer models and leverages the locality observed both within and between virtual address ranges. Although the sequential accesses in VABlocks from the previous section may suggest a straightforward contiguous prefetching approach instead of employing a tree structure for memory management, we prefer the tree structure for its broader applicability. UVM was originally designed for general-purpose use, not exclusively for machine learning. For instance, prefetching within the VABlock size under memory oversubscription provides significant performance gains for applications such as ATAX [32], BICG [32], and NW [33], which leverage the tree structure [16]. To ensure our approach benefits both large deep learning models and other general-purpose applications using UVM, maintaining the tree structure is essential for wider applicability.

Upon the first detection of a page fault from a VABlock, our block-aware prefetcher starts migrating contiguous VABlocks. This initial page fault indicates that all pages in the VABlock are stored in host memory, with none available in GPU memory. After the page fault is detected, our prefetcher initiates block-aware prefetching, proceeding through the three steps depicted in Fig. 8. The prefetching of a single adjacent VABlock is shown in Fig. 8(a), while Fig. 8(b) demonstrates prefetching multiple adjacent VABlocks, specifically two in this instance.

First, the UVM driver identifies the VABlocks adjacent to the initially faulted VABlock (❶). This is done by inserting prefetch entries while fetching page fault entries from the GPU fault buffer. Note that


UVM
CPU Virtual Address Space
GPU Virtual Address Space
Virtual Address Range
❷
2MB
❸
Page Fault
Initially Faulted VABlock
Prefetch Entry (❶)
Adjacent VABlock

(a) Prefetching a single adjacent VABlock.



(b) Prefetching two adjacent VABlocks.

Fig. 8. Operation of block-aware prefetcher.

a page fault entry refers to a faulted page that contains the requested data but is absent from the GPU memory, whereas a prefetch entry is an inserted entry that later increments the virtual address of the page fault by 2 MB. Upon fetching a page fault entry from the GPU fault buffer, the UVM driver first parses the entry into its respective page fault information fields. This information is then recognized by the UVM driver and incorporated into a fault cache entry structure. During this process, the corresponding virtual address for the page fault is identified, prompting the UVM driver to create a second entry with a virtual address increased by 2 MB from the previous entry, which is the initial page fault entry of the fault batch. This is accomplished by employing the left shift instruction as demonstrated in Algorithm 1. Shifting n bits to the left effectively multiplies the value by 2^n; thus, when a value shifted 21 bits to the left is added to the previous address, it results in a 2 MB incremented address. Additionally, other information fields, including the resident memory (*i.e.,* system memory), are inherited from the previous page fault entry to ensure the prefetch entry functions like a genuine page fault entry retrieved from the GPU fault buffer. Therefore, the first prefetch entry is generated with a 2 MB increment from the virtual address of the initial page fault, and subsequent prefetch entries follow this pattern by incrementing from the previous prefetch entry. These entries serve as page faults that trigger the generation of each adjacent VABlock, as the logically divided 2 MB VABlocks composing the virtual address space can be mapped to physical memory only upon access. To prefetch multiple VABlocks, neighboring logical VABlocks must be accessed, a process enabled by the prefetch entries in our proposed scheme. Thus, while the initial VABlock corresponds to the one containing the original fault, the adjacent VABlocks are generated by the inserted prefetch entries and are subsequently prefetched and migrated together with the initial VABlock.

Second, the initially faulted VABlock along with its adjacent VABlocks are created (❷). While logical 2 MB VABlocks signify segments divided by 2 MB in a virtual address range, the UVM driver creates a VABlock only when the pages within a logical VABlock are accessed. Thus, a prefetch entry with a 2 MB extended address operates as a regular page fault entry, allowing the UVM driver to create the corresponding VABlock containing the prefetch entry. During the initial phase of this process, the entries in the host-side fault cache are preprocessed for effective grouping into 2 MB VABlocks. The entries are first sorted by virtual address space. Each original page fault entry obtained from the GPU fault buffer is accompanied by its respective virtual address space information. However, the prefetch entries lack such details. Consequently, the UVM driver duplicates the virtual address space information from the first page fault entry to the subsequent prefetch entries. Then, the entries are sorted by virtual address. Since prefetch entries contain 2 MB incremented virtual addresses, they are relocated to the end of the array, with each one initiating the creation of a corresponding VABlock covering its virtual address. In order to create a 2 MB VABlock with a single prefetch entry, the prefetch threshold of the TBN prefetcher is configured to 1%. Additionally, we prevent the generation of VABlocks when the virtual addresses of prefetch entries exceed the virtual address range limit.

Third, the UVM driver migrates the created VABlocks to GPU memory (❸). Once the VABlocks are created by the initial page fault entry and the prefetch entries, they are then migrated altogether to the GPU memory. As described in Section 3.3, each VABlocks within a virtual address range are generally accessed sequentially. Therefore, migrating the VABlocks following the faulted VABlock helps to prevent potential page faults from those VABlocks, thereby saving time that would otherwise be spent handling them. Detailed results of the reduced page fault count are presented in Section 5.4.

The pseudocode for our proposed strategy is outlined in Algorithm 1, which demonstrates the block-aware prefetching of n VABlocks. The block-aware prefetcher inserts prefetch entries only when the fault index $i = 0$. Note that the fault index i refers to the position of a page fault entry within the current fault batch, with $i = 0$ representing the first page fault entry in the fault batch. By configuring the prefetch entries to be inserted when the fault index is 0, the block-aware prefetcher can prefetch adjacent VABlocks, even if the batch contains only a single page fault. With t acting as a temporary counter for the number of prefetched VABlocks, when prefetching two adjacent VABlocks, $t = 1$ in the first loop, and the fault address of the prefetch entry is calculated by adding 2 MB (1 × 2 MB) to the fault address. In

Algorithm 1 Block-Aware Prefetcher Mechanism

```
Input:  fault index $i$,
        prefetched VABlocks count $n$
Output: $(1 + n)$ VABlocks migrated to GPU memory

while $(i \le 256)$ & $(timestamp \le 30\ ns)$ & (pending page faults in GPU) do
    Fetch a page fault from GPU fault buffer
    if $i$ equals 0 then                                  ▷ Insert prefetch entries
        for $t = 1 \rightarrow$ n do
            $i = i + 1$
            fault address = previous fault address + $(1ULL << 21) * t$
        Goto next fault
for $i = 0 \rightarrow n$ do
    if fault address $\in$ virtual address range then
        Create VABlock
        Compute prefetch region per VABlock
    Migrate VABlocks to GPU Memory
Update GPU page tables
```

the second loop, where $t = 2$, the prefetch entry for the second VABlock adds 4 MB (2 × 2 MB) to the original fault address. Finally, after inserting n prefetch entries and creating the corresponding n VABlocks, only those within the virtual address range boundary persist and are migrated along with the initially faulted VABlock.

The prefetch strategies discussed in prior works strictly limit the migration scope to a single 2 MB VABlock. In contrast, our block-aware prefetcher enables the UVM driver to prefetch neighboring VABlocks, provided they all fall within the same virtual address range. As discussed in Section 2.3, the number of VABlocks within a single virtual address range can vary significantly, ranging from a few to hundreds, depending on the execution environment. Therefore, an important consideration is determining the range (or number) of VABlocks to prefetch alongside the VABlock that initially triggered the page fault. Based on our sensitivity study, detailed in Section 5.3, we discovered that prefetching 16 contiguous VABlocks yields the best performance. Consequently, in our proposed block-aware prefetcher, we set the number of prefetched blocks to 16 contiguous VABlocks, which leads to a total migration of 17 VABlocks, including the initially faulted VABlock.

5. Evaluation

In this section, we evaluate the block-aware prefetcher and compare its performance with other prefetching strategies. Whereas the TBN prefetcher exploits the locality within a VABlock, our proposed block-aware prefetcher expands this concept by prefetching multiple contiguous VABlocks to capture a wider scope of locality. We also present a sensitivity study on various numbers of prefetched VABlocks.

5.1. Methodology

Our proposed scheme is primarily designed to accelerate inference in large deep learning models, with all experiments focused on inference tasks for each application. However, it is also applicable to training, as the forward pass in training closely resembles that of inference. The implementation of our prefetcher and its performance effects during training are detailed in Section 5.5. All experiments in this work are performed on NVIDIA RTX 3090 and A6000 GPUs with an open-source NVIDIA UVM driver (Version 545.29.06) [34] and CUDA

Table 2
System configuration of experimental environment.

System configuration		
CPU	AMD Ryzen Threadripper 3960x	
System memory	DDR4-3200 256 GB	
Interconnect	PCIe 16x	
OS	Ubuntu Server 18.04.6 LTS	
GPU specification		
Model	NVIDIA RTX 3090	NVIDIA RTX A6000
GPU memory	GDDR6X 24 GB	GDDR6X 48 GB
Driver version	545.29.06	545.29.06
CUDA version	11.0	11.0

Table 3
Benchmarks used in experiments.

Model	Number of parameters	Memory Oversub. in RTX 3090	Memory Oversub. in RTX A6000
GPT-2 [36]	355M	×	×
	774M	×	×
	1.5B	○	×
GPT-3 [37]	6.7B	○	○
	13B	○	○

11.0 [35]. The UVM driver is modified to implement the block-aware prefetching strategy. The detailed system configuration is provided in Table 2.

To evaluate the performance of the block-aware prefetcher, we use the applications listed in Table 3. We execute the open-source GPT model in different sizes, ranging from 355M to 13B, on real hardware to demonstrate both cases of memory oversubscription and cases where the memory footprint remains within the available GPU memory (referred to as undersubscription) [15]. The performance of each model is evaluated by averaging the results from 100 executions. The second-to-last column of Table 3 specifies whether each application exceeds the GPU memory capacity on an RTX 3090, while the rightmost column indicates the same for an RTX A6000.

To integrate the functionalities of UVM into Transformer models, we build a UVM-PyTorch environment by adjusting the standard PyTorch (Version 1.8.0) [38]. This requires replacing `cudaMalloc()` with `cudaMallocManaged()`, which allocates memory that is automatically managed by UVM. Thus, the adapted deep learning framework UVM-PyTorch facilitates automatic page migration for applications, based on the demand paging scheme of UVM, without requiring user intervention. Although it is theoretically possible to handle memory oversubscription by manually managing data eviction and migration using `cudaMemAdvise()` and `cudaMemPrefetchAsync()`, our efforts to locate open-source implementations capable of executing GPT models in this manner without OOM errors were unsuccessful. Consequently, while tailored prefetching methods are feasible, they impose significant limitations, reinforcing UVM as the more flexible and efficient solution.

5.2. Performance analysis

Fig. 9 presents the performance comparison of the proposed block-aware prefetcher with vanilla PyTorch without UVM, the TBN prefetcher with 51% and 1% prefetch thresholds, the increased batch approach [19], and Early-Adaptor (EA) [16], for inference across the five applications listed in Table 3, on the RTX 3090. Performance is normalized against the TBN prefetcher with the default prefetch threshold of 51% on the RTX 3090, which represents the baseline approach set by default in the UVM driver. For the block-aware prefetcher, we configure the number of prefetched VABlocks to 16. Detailed results of the sensitivity study on the number of prefetched VABlocks are evaluated in Section 5.3. Vanilla PyTorch, which uses the CTE programming model without UVM, can only support applications

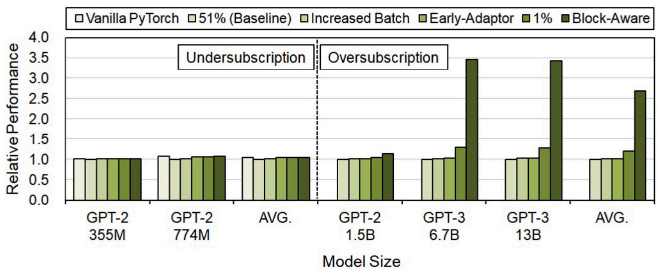


Fig. 9. Performance comparison of different prefetching strategies. Performance is normalized to the baseline, which is the default TBN prefetcher with a 51% threshold on the RTX 3090 for each model size. The vanilla PyTorch configuration refers to the CTE programming model without UVM. Increased batch and Early-Adaptor (EA) are two different page management strategies from related work. The 1% configuration represents the TBN prefetcher with an aggressive 1% threshold. The performance of block-aware prefetcher is evaluated with 16 VABlocks prefetched each round.

that fit within GPU memory, such as the GPT-2 355M and 774M models. The TBN prefetcher is tested with two configurations, one with the default prefetch threshold of 51% and another with a more aggressive 1% threshold. For the aggressive configuration, the entire faulted VABlock is prefetched even for a single page fault, as detailed in Section 2.3. The increased batch approach employs a larger fault batch size to handle more page faults at once, thereby reducing the total number of fault batches to be serviced [19]. The EA framework dynamically adjusts the prefetch threshold of the TBN prefetcher, based on the page fault history [16]. Finally, our block-aware prefetcher exceeds the 2 MB limit imposed by the TBN prefetcher and migrates multiple adjacent VABlocks, as explained in Section 4.

As mentioned earlier, the leftmost two applications, GPT-2 355M and 774M models, represent cases where GPU memory is not oversubscribed. In these cases, vanilla PyTorch, the increased batch approach, the EA framework, the aggressive TBN prefetcher with a 1% prefetch threshold, and the block-aware prefetcher achieve performance gains of 4.2%, 1.1%, 3.4%, 3.4%, and 4.0%, respectively, compared to the baseline. When memory is undersubscribed, eviction does not occur, and page migration frequency remains low, resulting in minimal differences in performance across the various memory management strategies. While the conventional CTE programming model (*i.e.,* using vanilla PyTorch) generally outperforms UVM, as the demand paging scheme of UVM loads data into GPU memory only upon access, it is noteworthy that our block-aware prefetcher achieves comparable performance to the CTE model.

On the other hand, the rightmost three applications, GPT-2 1.5B, GPT-3 6.7B, and GPT-3 13B models, represent cases where GPU memory is oversubscribed. These applications cannot be executed on vanilla PyTorch without UVM due to OOM issues. In these cases, the increased batch approach, the EA framework, the aggressive TBN prefetcher with a 1% prefetch threshold, and the block-aware prefetcher show performance gains of 1.0%, 1.6%, 1.2x and 2.7x, respectively, compared to the baseline. Aggressive prefetching shows clear performance improvements over the baseline by exploiting the spatio-temporal locality between pages in Transformer applications. Our block-aware prefetcher takes this approach further with an even more aggressive prefetching strategy, exceeding the performance of the TBN prefetcher. Additionally, aggressive prefetching has a greater impact on performance for larger models that oversubscribe GPU memory. As computation increases, page migrations and evictions become more frequent, highlighting the need for rapid prefetching in larger segments. Consequently, among the three applications, larger models tend to experience more significant performance gains due to the higher efficacy of aggressive prefetching.

The increased batch approach [19] proposed by Kim et al. results in an average GPU performance improvement of 1.0%. This modest gain is

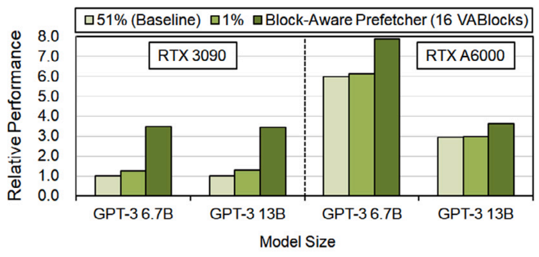


Fig. 10. Performance comparison of block-aware prefetcher across RTX 3090 and RTX A6000 GPUs. Performance is normalized to the baseline, which is the default TBN prefetcher with a 51% threshold on the RTX 3090 for each model size. The 1% configuration represents the TBN prefetcher with an aggressive 1% threshold. The performance of block-aware prefetcher is evaluated with 16 VABlocks prefetched each round.

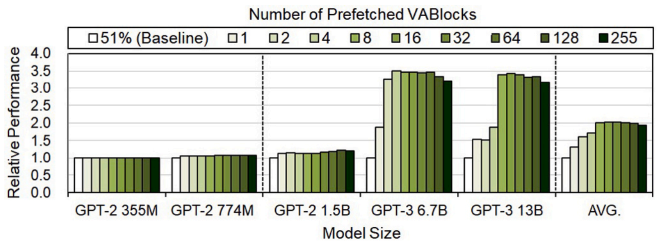


Fig. 11. Performance comparison of various numbers of prefetched VABlocks. Performance is normalized to the baseline, which is the default TBN prefetcher at a 51% threshold on the RTX 3090 for each model size. The configuration from 1 to 255 represents the number of prefetched VABlocks using the proposed block-aware prefetcher.

primarily attributed to the characteristics of Transformer applications. While the default fault batch size is set to 1 MB (256 page faults × 4 kB), our experiments across batch sizes ranging from 1 MB to 25 MB demonstrate that a 2 MB fault batch size yields the best performance. However, Transformer applications generate page faults from the same VABlock at once, followed by an interval before faults from a different VABlock are generated. The interval results in a timeout due to the 30 ns time limit that the driver waits for valid page fault entries while fetching page faults from device to host. This prevents the driver from fetching more page faults, even with a larger batch size, so we removed the time limit to fix this issue. This adjustment, however, increases the time required to service the first page fault, mainly due to the consistent intervals between memory accesses for different VABlocks, resulting in marginal performance gains.

The EA framework [16] proposed by Go et al. improves GPU performance by an average of 2.4%. This improvement is significantly lower than the 19.8% (*i.e.,* 1.2x) performance gain provided by the TBN prefetcher with a 1% threshold. This stems from the EA framework's strategy of initially disabling prefetching when memory oversubscription is detected and its cautious approach of gradually reducing the prefetch threshold. Since the framework raises the threshold to 100% when GPU memory is oversubscribed and reduces it by 25% at a time based on the page fault rate of the VABlock, it results in lower performance than the more aggressive approach of prefetching with a fixed 1% threshold.

In contrast, our block-aware prefetcher enhances GPU performance through increased aggressiveness by prefetching multiple adjacent VABlocks along with the faulted VABlock. Aggressive prefetching outperforms the baseline for Transformer models, leveraging the spatio-temporal locality between pages described in Section 3.3. By scaling this locality further, our block-aware prefetcher enhances GPU performance through a more aggressive prefetching approach, surpassing the TBN prefetcher across all five model sizes. It also outperforms previous memory management optimization approaches in UVM, including the increased batch approach and the EA framework. By prefetching a larger region of pages without altering the UVM driver's fault fetching time limit, our block-aware prefetcher delivers a 2.6x performance improvement compared to the increased batch size approach. Also, while the EA framework only prefetches pages within a single VABlock, our block-aware prefetcher increases prefetch aggressiveness by fetching multiple VABlocks, achieving a 2.6x performance improvement over the EA framework.

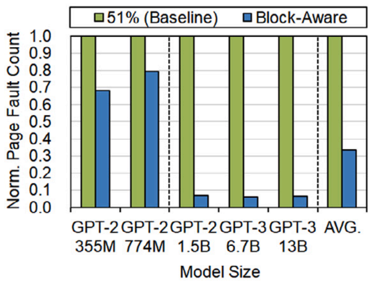
Fig. 10 compares the relative performance of the default TBN prefetcher with the baseline threshold of 51% and an aggressive 1%, as well as our block-aware prefetcher (16 VABlocks per round) on both the RTX 3090 and A6000. Performance is normalized to the TBN prefetcher with the baseline 51% threshold on the RTX 3090 for each application. Note that both the GPT-3 6.7B and 13B models experience memory oversubscription on the RTX A6000, as shown in Table 3. On the RTX A6000, the baseline performance of the 6.7B and 13B models improves by 6.0x and 2.9x, respectively, driven by the increased device memory size (48 GB on the RTX A6000 vs. 24 GB on the RTX 3090). The TBN prefetcher with an aggressive prefetch threshold of 1% shows performance gains of 2.3% and 1.6%, respectively, compared to the baseline threshold of 51%, on the RTX A6000, as aggressive prefetching proves to be effective for Transformer models. Furthermore, block-aware prefetching achieves 1.3x and 1.2x improvements over the baseline for the 6.7B and 13B models, respectively, on the RTX A6000. These results further demonstrate that our block-aware prefetcher provides consistent performance improvements across different GPU systems, emphasizing its flexibility and compatibility.

5.3. VABlock sensitivity study

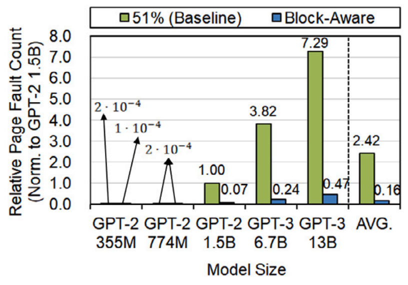
Our proposed block-aware prefetcher exploits the locality both within and between virtual address ranges, a feature that the default TBN prefetcher fails to address. Based on this locality, the block-aware prefetcher migrates multiple contiguous VABlocks adjacent to the faulted VABlock. A key factor here is the optimal number of VABlocks to prefetch.

Fig. 11 illustrates the performance of the block-aware prefetcher with different numbers of prefetched VABlocks, normalized to the performance of the TBN prefetcher with the baseline prefetch threshold of 51% on the RTX 3090. We evaluate the performance of the block-aware prefetcher when prefetching 1, 2, 4, 8, 16, 32, 64, 128, and 255 adjacent VABlocks. Since a fault batch can accommodate up to 256 page faults, the maximum number of prefetch entries allowed in a fault batch is 255. This is because the block-aware prefetcher must first encounter at least one page fault to incorporate additional prefetch entries. Therefore, a maximum of 255 VABlocks can be prefetched at once, allowing us to evaluate performance of up to 255 prefetched VABlocks.

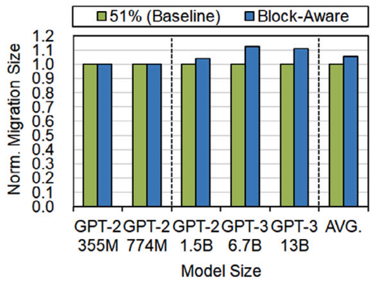
Among the five applications listed in Table 3, prefetching 1, 2, 4, 8, 16, 32, 64, 128, and 255 VABlocks show performance gains of 1.3x, 1.6x, 1.7x, 2.0x, 2.1x, 2.0x, 2.0x, 2.0x, and 1.9x, respectively, compared to the default TBN prefetcher with a 51% threshold. In general, performance improves as more VABlocks are prefetched, as this prevents additional page faults in those areas. The highest performance is obtained by prefetching 16 VABlocks. Beyond this point, performance gains tend to saturate with additional prefetching, influenced by the distinct sizes of virtual address ranges. Since a virtual address range is a contiguous bound of memory allocation, it has a unique size, resulting in various numbers of VABlocks for each range, from just a few to hundreds. When prefetching multiple VABlocks, the block-aware prefetcher stops once the address of a prefetch entry exceeds the limit of the corresponding virtual address range. Therefore, performance may become similar beyond a certain point (*i.e.,* 16 VABlocks), regardless of the number of prefetched VABlocks, due to invalid VABlocks that are discarded during processing. For instance, for a virtual address range



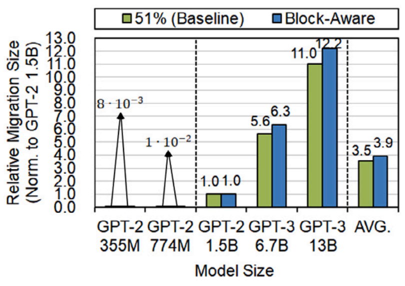
(a) Comparison of page fault count.



(b) Comparison of page fault count normalized to GPT-2 1.5B model.



(c) Comparison of migration size.



(d) Comparison of migration size normalized to GPT-2 1.5B model.

Fig. 12. Comparison of total page fault count and migration size. The left figures are normalized to the baseline performance, which is the TBN prefetcher with a 51% threshold on the RTX 3090 for each model size. The right figures are normalized to the baseline performance of the GPT-2 1.5B model, across all model sizes. The performance of block-aware prefetcher is evaluated with 16 VABlocks prefetched each round.

containing 10 VABlocks, prefetching 16 or 128 VABlocks yields similar performance, as both cases allow for only 9 VABlocks to be prefetched, following a page fault from the first VABlock.

5.4. Page migration statistics

Using the block-aware prefetcher enhances GPU performance by reducing the total number of page faults. For instance, prefetching 16 adjacent VABlocks helps to prevent page faults within the following 32 MB (16 × 2 MB) area. Fig. 12 shows the decrease in page faults and the increase in migration size for the five applications listed in Table 3, resulting from the block-aware prefetching of 16 VABlocks.

Figs. 12(a) and 12(b) illustrate the number of page faults that occurred during executing each application. While Fig. 12(a) is normalized to the TBN prefetcher with a baseline 51% threshold on the RTX 3090, Fig. 12(b) is normalized to the page fault count of the GPT-2 1.5B model using the baseline TBN prefetcher with a 51% threshold. As shown in Fig. 12(a), the block-aware prefetcher shows an average reduction of 66.7% in page faults compared to the baseline. Specifically, smaller models that fit within GPU memory capacity exhibit an average reduction of 26.4%, while larger models that exceed memory limits demonstrate a notable 93.5% decrease. This reduction in page faults results from the enlarged prefetched regions that prevent page faults from arising in those areas. Also, larger models experience a much steeper decline, as their high computational demands make them more sensitive to prefetching larger chunks of data.

While Fig. 12(a) presents the page fault count normalized to the baseline for each model size, Fig. 12(b) shows the relative page fault count normalized to the GPT-2 1.5B model. Fig. 12(b) demonstrates that our block-aware prefetcher reduces page faults by 15.9x (from 3.82 to 0.24) for the GPT-3 6.7B model and by 15.5x (from 7.29 to 0.47) for the GPT-3 13B model. Both models demonstrate a similar reduction in page faults, which explains the comparable performance observed in Fig. 9.

Fig. 12(c) presents the total host-to-device migration size for each model, while Fig. 12(d) presents the relative migration size normalized to that of the GPT-2 1.5B model. While Fig. 12(c) is normalized to the TBN prefetcher with a baseline 51% threshold on the RTX 3090, Fig. 12(d) is normalized to the migration size of the GPT-2 1.5B model using the baseline TBN prefetcher with a 51% threshold. In Fig. 12(c), the block-aware prefetcher shows a 5.6% increase on average, compared to the baseline. For smaller models, such as GPT-2 355M and 774M models, the migration size aligns with the baseline, as they remain within GPU memory limits. In contrast, larger models that exceed GPU memory capacity experience an increase in total migration size. While the baseline TBN prefetcher is limited to prefetching at most one VABlock at a time, the block-aware prefetcher requires a greater volume of pages to be transferred from host to device. Consequently, models employing the block-aware prefetcher result in a greater migration size, as an entire set of 16 VABlocks is migrated each time a page fault occurs. Additionally, Fig. 12(d) shows that the migration size for the GPT-3 13B model is 1.9x larger than that of the GPT-3 6.7B model, which aligns with Fig. 12(b), indicating that the GPT-3 13B model experiences 1.9x more page faults than the 6.7B model.

The increase in migration size shown in Fig. 12(c) also suggests that block-aware prefetching takes more time to handle page faults than the default TBN prefetcher, due to the larger volume of pages being migrated into the GPU. Based on our analysis, block-aware prefetching of 16 VABlocks takes, on average, 5.3x longer to fetch and replay page faults than the baseline. However, it is important to note that this increased handling time is approximately 66.9% shorter than fetching 16 VABlocks individually, which would involve 16 separate page fault handling processes. Thus, while the time to handle page faults increases due to the larger area being prefetched, this increase is minimal compared to the significantly longer time the baseline TBN prefetcher would take to process the same area one VABlock at a time. Despite the overall increase in migration size and fault handling time, the block-aware prefetcher significantly reduces the total number of page faults by loading large sets of VABlocks into GPU memory, ultimately improving GPU performance, as discussed in Section 5.2.

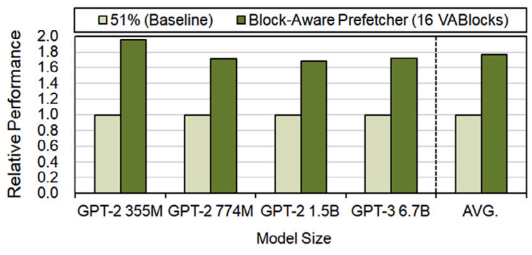


Fig. 13. Performance comparison of block-aware prefetcher for training tasks with two RTX 3090 GPUs. Performance is normalized to the baseline, which is the TBN prefetcher with a 51% threshold on two RTX 3090 GPUs for each model size. The performance of block-aware prefetcher is evaluated with 16 VABlocks prefetched each round.

5.5. *Training performance and discussion*

While our paper primarily focuses on inference tasks, our block-aware prefetcher is also effective for training tasks, as training involves forward computation steps similar to those in inference. It is widely recognized that training is often performed using multiple GPUs due to its high computational demands. Our block-aware prefetcher can be seamlessly integrated into such multi-GPU environments without any algorithmic adjustments. As detailed in Section 4, prefetch entries are inserted during the fault fetching process, relying on the host's ability to identify the faulting GPU. In a multi-GPU setup, the host continues to identify the faulting GPU, and the fault fetching process is triggered independently on each GPU, with each device interacting directly with the host.

We conduct an additional experiment by applying our block-aware prefetcher in a multi-GPU setup with two RTX 3090 GPUs. Training requires more computational resources than inference, as it involves calculating the loss and gradients, which are necessary for adjusting the model weights. This additional computation, part of the backward pass, also demands extra memory to store intermediate values. Given the heavy computational demands of training, we use four out of the five model sizes listed in Table 3, ranging from 355M to 6.7B parameters, since the GPT-3 13B model exceeds the system memory capacity and causes OOM errors. Moreover, of the four trainable model sizes, all but the GPT-2 355M model lead to GPU memory oversubscription. This indicates that memory oversubscription is significantly more problematic during training than during inference, where it begins with the GPT-2 1.5B model. As the memory access patterns for different model sizes are similar during inference, we expect the same behavior during training, implying that the GPT-3 13B model will perform similarly to other model sizes. As training large Transformer models with limited resources is more challenging than inference, UVM proves to be an ideal solution for training deep learning models across multiple GPUs.

Fig. 13 presents the performance of the block-aware prefetcher with 16 VABlocks prefetched, measured across training tasks for various model sizes. The performance is normalized against the baseline TBN prefetcher with a 51% threshold on two RTX 3090 GPUs. Our block-aware prefetcher achieves an average performance improvement of 1.8x over the baseline in training tasks. We employ DistributedDataParallel for multi-GPU parallelism, which is the standard parallelism method supported in PyTorch [39]. As the forward pass and inference share similar memory access patterns, our prefetcher enhances performance during the forward pass. However, the backward pass may involve different memory access patterns, resulting in a more limited performance improvement during that phase. In future work, we plan to explore these memory access patterns during the backward pass to achieve greater performance improvements, complementing those made in the forward pass.

6. Related work

Prior studies have explored various issues related to UVM, leading to the development of strategies for effective page management. Zhu et al. focus on resolving the issue of redundant memory transfers caused by the automatic data transfer feature of the demand paging scheme in UVM [40]. They propose a novel memory discard directive that enables users to inform the UVM system when the data within a specified virtual address range is no longer needed, leading to the page being discarded and reducing unnecessary data transfers. Haque Rafi et al. address the problem of false sharing, which occurs when both the CPU and GPU access different regions of the same page, resulting in unnecessary data migration between the two [41]. They develop a tool to detect false sharing by monitoring the data access patterns of both the CPU and GPU, which helps define the access boundaries for both processors. Kim et al. propose increasing the fault batch size to amortize the high costs in page fault handling [19]. By increasing the fault batch size, more page faults can be processed at once, reducing the number of batches that need to be managed. Since page fault handling introduces latency, reducing these processes helps enhance GPU performance.

Several works have analyzed the performance of UVM by observing memory accesses across various workloads; however, none have specifically targeted Transformer models with a real-time analysis. Ganguly et al. present an advanced runtime that detects the patterns within CPU–GPU interconnect traffic [42]. This is done by examining the patterns in page faults, which results in the effective analysis of memory access behaviors. They categorize GPU applications into two groups: *regular* applications with dense, sequential, and repetitive memory access and *irregular* applications with sparse and seldom memory access. Similarly, Allen and Ge demonstrate a quantitative evaluation of page fault handling, incorporating diverse cases of prefetching and oversubscription [30]. To achieve this, they examine the spatial locality of pages by observing the distribution of page faults per fault batch. In a separate study, they conduct an extensive analysis on the overall process of page fault handling of UVM, identifying key cost components by investigating how faults are generated and managed by the UVM driver [17]. Shao et al. identify the underlying causes of performance decline in GPU applications under memory oversubscription [43]. They investigate the diverse sensitivities to oversubscription through measuring performance with different oversubscription ratio and tracking page fault patterns. These UVM-related studies are performed by integrating UVM functionalities into standard benchmarks, particularly through the replacement of `cudaMalloc()` with `cudaMallocManaged()`, as mentioned in Section 5.1. Gu et al. provide a UVM benchmark suite of 32 representative benchmarks from a wide range of application domains [44]. This benchmark consists of four machine learning workloads, all of which are modified to the CUDA programming language to leverage `cudaMallocManaged()`. Our work, however, adopts a more authentic strategy by developing UVM-compatible PyTorch, making it applicable to a wider range of deep learning workloads.

Prefetching and pre-evicting data are both key topics for effective memory management in UVM. While these techniques are typically operated within a 2 MB limit, our research is the first to break free from this constraint. Go et al. provide a real-time analysis of a dynamic prefetcher that automatically controls the prefetch aggressiveness according to the page access patterns of applications [16]. By monitoring page faults and calculating the page fault ratio, the prefetcher evaluates whether to prefetch aggressively or selectively based on the observations. Ganguly et al. address the limitations of eviction policies that overlook prefetch methods, by proposing pre-eviction policies compatible with the current prefetch policies [10]. The proposed pre-eviction policies utilize either LRU or locality information from the TBN prefetcher. Yu et al. incorporate a page set chain into the GPU driver to manage referenced pages based on the order of LRU and MRU [45]. They further suggest hierarchical page eviction that dynamically selects the appropriate eviction strategy for the executing application. However, this policy proves inefficient when combined with prefetching, since it relies on a page set counter that counts accessed pages to detect access patterns, which can be disrupted by counting prefetched pages. Therefore, Yu et al. propose a modified page eviction policy and a memory access pattern-aware prefetcher in alignment with the eviction policy [46]. The modified eviction policy considers prefetch semantics, allowing the prefetcher to fetch pages based on access patterns observed in eviction candidates, ensuring a cohesive interaction between them. However, these works differ from our work by using a simulation framework for its experiments, resulting in findings that are validated only to a limited extent, as they lack real-time analysis. While the prefetch and pre-eviction schemes proposed in these works strictly limit the migration scope to a single VABlock, our work introduces the first prefetcher to extend beyond the limited 2 MB boundary.

7. Conclusion

This paper explores the effects of prefetching aggressiveness on GPU performance, analyzing the effect of different prefetch thresholds and the memory access patterns observed during inference in Transformer models. Our experiments reveal that aggressive prefetching proves effective across various model sizes, particularly benefiting larger models that exceed GPU memory limits. Further analysis shows that this effectiveness arises from the spatio-temporal locality between adjacent pages. While the default TBN prefetcher primarily exploits this locality, our analysis highlights a broader locality based on the contiguity in memory accesses within and between virtual address ranges. Based on these observations, we propose a block-aware prefetcher, designed to concurrently migrate multiple VABlocks adjacent to the faulted VABlock, thereby preventing additional page faults from occurring in those areas. This mechanism takes full advantage of sequential VABlock accesses and the contiguity across virtual address ranges. Our experiments on real hardware show that the proposed block-aware prefetcher achieves an average performance improvement of 2.7x compared to the baseline TBN prefetcher, under memory oversubscription.

CRediT authorship contribution statement

Jane Rhee: Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Ikyoung Choi:** Validation, Software, Conceptualization. **Gunjae Koo:** Writing – original draft, Methodology, Conceptualization. **Yunho Oh:** Writing – original draft, Supervision, Methodology, Conceptualization. **Myung Kuk Yoon:** Writing – original draft, Supervision, Resources, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Myung Kuk Yoon reports financial support was provided by Institute for Information & Communications Technology Planning & Evaluation. The corresponding author collaborates with professors from Yonsei University and Korea University in South Korea, as well as the University of Illinois Urbana-Champaign in the USA. Yunho Oh was previously employed by Sungkyunkwan University. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-00155966, Artificial Intelligence Convergence Innovation Human Resources Development (Ewha Womans University)) and by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) of the Government of South Korea (No. NRF-2021R1C1C1012172 and NRF-2022R1C1C1011021). Yunho Oh and Myung Kuk Yoon are the corresponding authors.

Data availability

The authors do not have permission to share data.

References

[1] Xi Chen, The advance of deep learning and attention mechanism, in: 2022 International Conference on Electronics and Devices, Computational Science, ICEDCS, 2022, pp. 318–321.

[2] Johannes Schneider, What comes after transformers? – A selective survey connecting ideas in deep learning, 2024, URL https://arxiv.org/abs/2408.00386. (Accessed 14 October 2024).

[3] Wilfried Bounsi, Borja Ibarz, Andrew Dudzik, Jessica B. Hamrick, Larisa Markeeva, Alex Vitvitskyi, Razvan Pascanu, Petar Veličković, Transformers meet neural algorithmic reasoners, 2024, URL https://arxiv.org/abs/2406.09308. (Accessed 14 October 2024).

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 6000–6010.

[5] Saritha Vinod, M Naveen, Asis K Patra, Anto Ajay Raj John, Accelerating towards larger deep learning models and datasets – A system platform view point, in: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2020, pp. 1–7.

[6] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, Joseph E. Gonzalez, Train large, then compress: Rethinking model size for efficient training and inference of transformers, in: Proceedings of the 37th International Conference on Machine Learning, 2020.

[7] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, Torsten Hoefler, Data movement is all you need: A case study on optimizing transformers, 2021, URL https://arxiv.org/abs/2007.00072. (Accessed 9 September 2024).

[8] NVIDIA, NVIDIA pascal architecture, 2016, URL https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (Accessed 21 August 2024).

[9] AMD, Radeon's next-generation vega architecture, 2017, URL https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf. (Accessed 21 August 2024).

[10] Debashis Ganguly, Ziyu Zhang, Jun Yang, Rami Melhem, Interplay between hardware prefetcher and page eviction policy in CPU-gpu unified virtual memory, in: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture, ISCA, 2019, pp. 224–235.

[11] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, Minyi Guo, Grus: Toward unified-memory-efficient high-performance graph processing on GPU, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 18 (2) (2021) 1–25.

[12] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, Yale N. Patt, Improving GPU performance via large warps and two-level warp scheduling, in: 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2011, pp. 308–317.

[13] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, Jun Yang, A framework for memory oversubscription management in graphics processing units, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 49–63.

[14] C. Hughes, Unified memory: GPGPU-sim/UVM smart integration, 2022, URL https://www.osti.gov/servlets/purl/1844477. (Accessed 9 October 2024).

[15] Yunmo Koo, Pytorch-UVM, 2020, URL https://github.com/kooyunmo/cuda-uvm-gpt2. (Accessed 14 September 2024).

[16] Seokjin Go, Hyunwuk Lee, Junsung Kim, Jiwon Lee, Myung Kuk Yoon, Won Woo Ro, Early-adaptor: An adaptive framework forproactive UVM memory management, in: 2023 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2023, pp. 248–258.

[17] Tyler Allen, Rong Ge, In-depth analyses of unified virtual memory system for GPU accelerated computing, in: SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–14.

[18] Mark Harris, Unified memory for CUDA beginners, 2017, URL https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. (Accessed 18 August 2024).

[19] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, Hyesoon Kim, Batch-aware unified memory management in GPUs for irregular workloads, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1357–1370.

[20] Nicolas Brunie, Caroline Collange, Gregory Diamos, Simultaneous branch and warp interweaving for sustained GPU performance, in: 2012 39th Annual International Symposium on Computer Architecture, ISCA, 2012, pp. 49–60.

[21] Prasanth Kolachina, Nicola Cancedda, Marc Dymetman, Sriram Venkatapathy, Prediction of learning curves in machine translation, in: Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1, 2012, pp. 22–30.

[22] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, Yanqi Zhou, Deep learning scaling is predictable, empirically, 2017, URL https://arxiv.org/abs/1712.00409. (Accessed 11 September 2024).

[23] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, Dario Amodei, Scaling laws for neural language models, 2020, URL https://arxiv.org/abs/2001.08361. (Accessed 11 September 2024).

[24] Andrew Brock, Jeff Donahue, Karen Simonyan, Large scale GAN training for high fidelity natural image synthesis, 2019, URL https://arxiv.org/abs/1809.11096. (Accessed 9 September 2024).

[25] A. Silva, F. Rocha, A. Santos, G. Ramalho, V. Teichrieb, GPU pathfinding optimization, in: 2011 Brazilian Symposium on Games and Digital Entertainment, 2011, pp. 158–163.

[26] Marko J. Mišić, Đorđe M. Đurđević, Milo V. Tomašević, Evolution and trends in GPU computing, in: 2012 Proceedings of the 35th International Convention MIPRO, 2012, pp. 289–294.

[27] NVIDIA, NVIDIA ampere GA102 GPU architecture, 2020, pp. 1–53, URL https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf. (Accessed 24 August 2024).

[28] NVIDIA, NVIDIA ampere GA102 GPU architecture, 2021, pp. 1–53, URL https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf. (Accessed 24 August 2024).

[29] Nurlan Nazaraliyev, Elaheh Sadredini, Nael Abu-Ghazaleh, GPUVM: GPU-driven unified virtual memory, 2024, URL https://arxiv.org/abs/2411.05309. (Accessed 31 December 2024).

[30] Tyler Allen, Rong Ge, Demystifying GPU uvm cost with deep runtime and workload analysis, in: 2021 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2021, pp. 141–150.

[31] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, Jeff Dean, Efficiently scaling transformer inference, in: Conference on Machine Learning and Systems, 2022.

[32] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, John Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: 2012 Innovative Parallel Computing, InPar, 2012, pp. 1–10.

[33] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, Kevin Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: 2009 IEEE International Symposium on Workload Characterization, IISWC, 2009, pp. 44–54.

[34] NVIDIA, NVIDIA linux open GPU kernel module source, 2022, URL https://github.com/NVIDIA/open-gpu-kernel-modules/. (Accessed 18 August 2024).

[35] NVIDIA, NVIDIA CUDA toolkit 11.0.3, 2020, pp. 1–27, URL https://docs.nvidia.com/cuda/archive/11.0/pdf/CUDA_Toolkit_Release_Notes.pdf. (Accessed 27 August 2024).

[36] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, Language models are unsupervised multitask learners, in: Conference on Machine Learning and Systems, 2019, URL https://api.semanticscholar.org/CorpusID:160025533.

[37] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei, Language models are few-shot learners, 2020, URL https://arxiv.org/abs/2005.14165. (Accessed 9 September 2024).

[38] PyTorch, Pytorch documentation, 2019, URL https://pytorch.org/docs/1.8.0/. (Accessed 28 September 2024).

[39] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, Soumith Chintala, Pytorch distributed: experiences on accelerating data parallel training, in: VLDB Endowment, 2020, pp. 3005–3018.

[40] Weixi Zhu, Guilherme Cox, Jan Vesely, Mark Hairgrove, Alan L. Cox, Scott Rixner, UVM discard: Eliminating redundant memory transfers for accelerators, in: 2022 IEEE International Symposium on Workload Characterization, IISWC, 2022, pp. 27–38.

[41] Md Erfanul Haque Rafi, Kaylee Williams, Apan Qasem, Raptor: Mitigating CPU-gpu false sharing under unified memory systems, in: 2022 IEEE 13th International Green and Sustainable Computing Conference, IGSC, 2022, pp. 1–8.

[42] Debashis Ganguly, Rami Melhem, Jun Yang, An adaptive framework for oversubscription management in CPU-GPU unified memory, in: 2021 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2021, pp. 1212–1217.

[43] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, Minyi Guo, Oversubscribing GPU unified virtual memory: Implications and suggestions, in: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, 2022, pp. 67–75.

[44] Yongbin Gu, Wenxuan Wu, Yunfan Li, Lizhong Chen, UVMBench: A comprehensive benchmark suite for researching unified virtual memory in GPUs, 2020.

[45] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Zhiying Wang, HPE: Hierarchical page eviction policy for unified memory in GPUs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (10) (2020) 2461–2474.

[46] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, Zhiying Wang, Coordinated page prefetch and eviction for memory oversubscription management in GPUs, in: 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2020, pp. 472–482.