An Intelligent Framework for Oversubscription Management in CPU-GPU Unified Memory

Xinjian Long · Xiangyang Gong · Bo Zhang · Huiyang Zhou

Received: 24 June 2022 / Accepted: 5 January 2023 / Published online: 14 February 2023
© The Author(s), under exclusive licence to Springer Nature B.V. 2023

Abstract Unified virtual memory (UVM) improves GPU programmability by enabling on-demand data movement between CPU memory and GPU memory. However, due to the limited capacity of GPU device memory, oversubscription overhead becomes a major performance bottleneck for data-intensive workloads running on GPUs with UVM. This paper proposes a novel framework for UVM oversubscription management in discrete CPU-GPU systems. It consists of an access pattern classifier followed by a pattern-specific transformer-based model using a novel loss function aiming to reduce page thrashing. A policy engine is designed to leverage the model’s result to perform accurate page prefetching and eviction. Our evaluation shows that our proposed framework significantly outperforms the state-of-the-art (SOTA) methods on a set of 11 memory-intensive benchmarks, reducing the number of pages thrashed by 64.4% under 125% memory oversubscription compared to the baseline, while the SOTA method reduces the number of pages thrashed by 17.3%. Compared to the SOTA method, our solution achieves average IPC improvement of 1.52X and 3.66X under 125% and 150% memory oversubscription.

Keywords Discrete CPU-GPU system · Unified virtual memory · Oversubscription · Deep learning

1 Introduction

Modern GPUs support an advanced feature called unified virtual memory (UVM [1]), which enables a unified virtual memory space and programmer-agnostic demand-driven automatic data migration between the CPU memory and the GPU device memory. UVM significantly enhances GPU programmability by relieving the developers of the data management burden.

Despite the attractive benefits, UVM also raises concerns about the efficiency of the GPU runtime’s data management strategy. Since there is only one physical copy of the data maintained in either the CPU memory or the GPU device memory, an inappropriate strategy may cause unnecessary memory transfers between the host and the GPU. As a result
of the relatively slow bandwidth of the CPU-GPU interconnect, these transfers may lead to a serious performance slowdown for the GPU workloads. Unfortunately, the limited capacity of the GPU device memory and the ever-growing size of the applications further increase the probability of the occurrence of this problem. Page thrashing unavoidably happens when the application’s working set size exceeds the device memory size. Currently, page thrashing becomes a first-order performance bottleneck for data-intensive applications using GPU UVM.

A few methods have been proposed to handle page thrashing in GPU UVM. Ganguly et al. leveraged the hardware-access counter and the zero-copy technique to propose an adaptive framework (UVMSmart [2]) for page migration and pinning to address the performance overhead of oversubscribed GPGPU workloads. Yu et al. proposed a hierarchical page eviction policy (HPE [3]) for GPU UVM. These works utilize knowledge extracted from in-depth analysis of inherent application characteristics (memory access patterns, etc.) to design specific rules for each workload. Although these methods significantly mitigate the performance impact of memory oversubscription, there are still problems left to be solved. First, a thorough prior understanding of workload memory access patterns in advance is not always possible in practice. A rule-based design for a particular subset of the GPGPU workloads may not be generic to the others. Second, memory access patterns may vary in different program phases. A simple combination of the existing data prefetchers and eviction policies cannot handle this variance. Third, existing mechanisms suffer from inefficiency when a data prefetcher and an eviction policy are combined.

To compensate for the weakness of the rule-based methods, machine intelligence is introduced to provide insights for improvement. Since there are no existing learning-based methods for oversubscription management in GPU UVM, we only discuss the learning-based works on other hardware prediction mechanisms in this paper. Hashemi et al. [4] apply the RNN model memory access patterns analysis, which demonstrates higher precision and recall than table-based approaches. Shi et al. applied deep learning to handle the cache replacement problem [5] and the data prefetching problem [6]. Existing learning-based works applied variant neural models and generated impressive performance improvement on different hardware issues by extracting knowledge from collected memory traces. However, unique challenges also emerge while attempting to make these learning-based methods practical for use. First, for the learning-based design that frames their issue as a classification problem, the number of classes may grow explosively in the running workload life cycle. Such a growing number of classes may cause the neural models to suffer from serious forgetting problems when they are continually updated with new incoming data. Second, some of the existing learning-based methods use an identical neural model to handle all the data, and some of the works create neural models for every unique page. Empirically, it is difficult for a single model to learn the knowledge from all the memory access patterns or all the GPGPU workloads, whereas creating too many neural models may require unacceptable computation and storage overhead.

To solve these challenges, we propose an intelligent framework for oversubscription management in CPU-GPU UVM in this paper. This framework takes as input a sequence of historical memory access information including page address, page delta, PC, and thread block ID. When the new data arrive, the input sequence is first fed to a memory access pattern classifier to identify which pattern it belongs to. According to the pattern classification results, a specific set of neural model weights is selected from a pattern-based model table and fed to a novel page predictor. In other words, each pattern’s input sequences are trained by a separate neural model. To extract knowledge from both regular (stride, constant, etc.) and irregular (pointer chase, etc.) memory access patterns, the page predictor is composed of two transformer-based [7] basic blocks to learn these patterns respectively. To solve the explosive growing number of classes problem, we introduce the incremental learning method into the page predictor to encourage the neural model to continue learning the new classes without forgetting the old classes. A novel loss function is used in the training of the page predictor, and this loss function helps the page predictor become thrashing-aware. All the prediction results within the same interval are aggregated and fed to a policy engine. By leveraging a prediction frequency table and a page set chain, the policy engine can learn the importance of different pages in the near future memory access of the workload. Then, the policy determines the prefetching or the eviction candidates according
to each page’s importance when the corresponding prefetching or eviction request arrives. Finally, the decision is sent to the GPU memory management unit (GMMU), and the corresponding memory operation is performed.

This paper makes the following contributions:

- We propose a transformer-based intelligent framework for oversubscription management in CPU-GPU unified virtual memory, which can significantly improve the GPU kernels’ performance under different levels of UVM oversubscription. As far as we know, it is the first work using deep learning for oversubscription management in CPU-GPU unified memory.

- We provide an in-depth analysis of the current rule-based methods for oversubscription management and the current learning-based methods on other computer architectural issues. We identify the necessity of applying machine intelligence for more accurate data prefetching and data eviction compared with rule-based methods, and we identify sources of performance loss for current learning-based methods while running in an online manner.

- We evaluate our proposed approach among 11 memory-intensive GPGPU benchmarks. Experimental results show that our solution achieves a 64.4% reduction on average in page thrashing compared to the baseline under 125% UVM oversubscription, while the SOTA work achieves a 17.3% reduction on average. Our solution achieves an average IPC improvement of 1.52X under 125% memory oversubscription, and our solution achieves an average IPC improvement of 3.66X under 150% memory oversubscription.

- We evaluate our proposed approach in the case of handling concurrent multiple GPGPU workloads, and we further evaluate our approach with machine learning workloads invoking the NVIDIA cuDNN APIs. Experimental results show that our design can be adapted to modern GPU architecture and provide performance improvement for real-world workloads.

The remainder of this paper is organized as follows. Section 2 presents the background of this work. Section 3 discusses the limitations of both the rule-based works on GPU memory oversubscription management and the learning-based works on other hardware prediction issues. Section 4 describes the design of our intelligent framework. Section 5 compares the results of our intelligent framework with both rule-based works and learning-based works. Section 6 discusses the related works of this paper before providing concluding remarks in Section 7.

2 Background

In this section, we review the general mechanics of on-demand paging in CPU-GPU UVM, the soft and hard pining, tree-based prefetchers, and page eviction mechanisms following the NVIDIA/CUDA terminology. It is worth noting that the techniques mentioned in this section as well as our design described in the following sections are adaptable to other GPU architectures in addition to NVIDIA.

2.1 On-Demand Page Migration and Soft/Hard Pinning

CPU-GPU UVM provides a single virtual address space accessible from both the CPU and GPU. Using CUDA, developers can apply UVM by calling the `cudaMallocManaged` API to allocate data that can be accessed by both host code and GPU kernels with a single shared pointer. The functionality of unified memory is enabled by on-demand memory allocation and fault-driven page migration. A miss in the last level translation lookaside buffer (TLB) is relayed to the GPU memory management unit (GMMU), which performs a page table walk for the requested page. If there is no page table entry (PTE) for the requested page or the valid flag is not set, then a far-fault is registered in the GMMU’s far-fault miss status handling registers (MSHR) and the corresponding warp will be stalled. Then this request is forwarded to the host and triggered a host-side page table walk. Once the page table walk is finished and the requested page is returned, MSHRs are consulted to notify the corresponding LDST to replay the device memory access, and then the stalled warp is marked executable. This
is demonstrated as sequence (2) in Fig. 1, and this is the general process of GPU UVM on-demand page migration.

Handling far-faults with on-demand migration is costly because of the high latency of page table walk and data migration over PCI-e interconnect. The NVIDIA CUDA runtime introduces pinning memory to alleviate this problem. On the one hand, developers can call the \texttt{cudaHostRegister} and the \texttt{cudaHostGetDevicePointer} APIs to force the memory allocation to be hard-pinned to the host memory. In this case, pages in such memory allocation are never transferred from host to device memory. GPU kernels can only request these pages using the remote direct memory access (RDMA). This is demonstrated as sequence (3) in Fig. 1, and this is the case of CUDA zero-copy. On the other hand, developers can call \texttt{cudaMemAdvise}, \texttt{cudaMemAdviseSetAccessedBy}, and \texttt{cudaMemAdviseSetPreferredLocation} APIs to advise the allocation to be soft-pinned to the host memory. In this case, pages in such allocation are not migrated to the device memory at the first touch. Rather, the migration is delayed until the number of read requests reaches a certain static threshold. This is demonstrated as the combination of (2) and (3) in Fig. 1. In addition, the IBM POWER9 processor’s address translation service (ATS [8]) enables the GPU to request translations using the CPU’s page tables rather than the GMMU. Changes in the CPU’s page table are immediately available to all processors without the OS needing to synchronize with the GPU. ATS supports delayed page migration between CPU and GPU memory similar to the aforementioned APIs.

2.2 Tree-Based Prefetcher

At the GPU Technology Conference 2018, a tree-based prefetcher was implemented by NVIDIA CUDA 8.0 driver. Ganguly et al. [9] uncovered the semantics of this tree-based neighborhood prefetcher through microbenchmarking and profiling. The user-requested size of a \texttt{cudaMallocManaged} allocation is logically divided into some 2 MB memory chunks plus a remainder. Each chunk is further logically divided into 64 KB basic blocks, which is the unit of prefetching. According to the far-faults received from the GPU, the runtime calculates the base addresses of the basic blocks corresponding to these faults. Then, these base addresses are sent to the IOMMU and all the pages within the corresponding basic blocks are migrated to the GPU. The runtime keeps track of the total size of valid memory residing in the GPU for each non-leaf node among all the 2 MB trees. If the runtime detects that any non-leaf node’s GPU valid memory is more than 50% of the total node capacity, the remaining non-valid pages of that node are scheduled as further prefetching candidates. Figure 2 illustrates such a tree structure for a 512 KB region. Tree-based Prefetcher, which migrates (virtually) contiguous pages around the first-touched page, not only mitigates the cost of demand paging but also helps reduce the TLB shootdown overhead. Because the GPU comprises many concurrently executing pipelines and the GPU uses a multi-level global page table, one TLB shootdown may require flushing all compute pipelines and it may stall thousands of execution lanes instead of one single core. By migrating pages that are likely to touch but have not yet touched, TLB invalidations/refills relative to these pages are not required compared to the other page migration policies which triggered the migration only after the page is touched [10].

2.3 Page Eviction

LFU is a representative frequency-based policy, but it is not sufficient for selecting an appropriate eviction policy for unified memory [3]. The widely-used recency-based policy LRU performs well for
An Intelligent Framework for Oversubscription...

Fig. 2 A tree-based neighborhood hardware prefetcher implemented by NVIDIA since CUDA 8.0 on a 512 KB memory chunk

A significant portion of applications. GPU Technology Conference 2017 [11] specified that the CUDA drivers implement the LRU page replacement policy. For unified memory, Zheng et al. [12] evaluated the performance of LRU and random for some applications. Ganguly et al. [9] introduced a tree-based eviction policy inspired by the tree-based prefetcher. At any instance, if the current occupancy of a non-leaf node falls below 50%, the runtime pre-evicts the other 64 KB valid leaf nodes under it. Yu et al. [3] proposed a hierarchical eviction policy that manages a page set chain dynamically and uses statistics to classify applications. It selects an appropriate eviction strategy based on the classification result.

3 Current Challenges

In this section, we describe the performance bottleneck associated with GPU UVM oversubscription. Then, we discuss the limitations of the existing rule-based methods for oversubscription management, and we discuss the limitations of the existing learning-based methods in computer architecture.

3.1 Oversubscription Overhead

Figure 3 shows that all GPGPU workloads suffer from performance loss due to memory oversubscription, and this loss intensifies as the oversubscription
level grows (the average performance slowdown under 125% oversubscription is 24.1%, and 3 of the applications (ATAX, NW, 2DCONV) are crashed under 150% oversubscription). More precisely, 125% oversubscription indicates that the device memory size equals 0.8 times the certain workload’s working set size. For instance, if the working set size of a certain application is 1 MB, then a device memory size of 0.8 MB will lead to 125% oversubscription.

The major cause of the performance loss of memory oversubscription (shown in Fig. 3) is page thrashing, which means that pages are moved back and forth between CPU and GPU memory repeatedly upon reaching GPU device memory capacity. In addition, the cooperation of inappropriate prefetching and eviction mechanisms may further worsen this problem. While useless pages are fetched in by aggressive prefetching, useful pages may be evicted since the capacity of the device memory is reached, and these pages will have to be brought back in the near future. In contrast, a recency-based eviction policy may mistakenly select the hot pages as eviction candidates because of the lack of information (frequency, pattern, etc.), which may cause instant thrashing and hurt the application performance. In the experiments shown in Fig. 3, the tree-based prefetcher and the LRU eviction policy are applied, which is the same as the CUDA runtime [1, 11].

3.2 Limitation of Existing Rule-Based Methods

Table 1 shows the number of pages thrashed using different strategies under 125% memory oversubscription. The baseline indicates the combination of the tree-based prefetcher and the LRU eviction policy as described in Section 3.1. Since Belady’s MIN algorithm [13] (MIN) provably minimizes the number of cache misses, it is considered optimal guidance for a data replacement problem. We assume all the data fetched in are demand load (D. or Demand.) and MIN is applied for page eviction upon the capacity of the memory being reached. We take this combination (D.+Belady.) as the theoretical upper bound of this study. HPE [3] and UVMSmart [2] are the two SOTA works in handling the memory oversubscription in GPU unified memory.

We can see that D.+Belady achieves the minimum page thrashed for all the GPU workloads among all the strategies. However, since Belady’s MIN algorithm is impractical and requires future knowledge to decide which page to evict. This result can only be guidance instead of a solution to this problem. Baseline causes the largest number of pages thrashed for most of the workloads. We believe that the reason for this result is because of the poor adaptability between the aggressive tree-based prefetching mechanism and the recency-based eviction policy as described in

| Benchmark   | Baseline | Demand.+HPE | UVMSmart | Demand.+Belady. |
|-------------|----------|-------------|----------|-----------------|
| AddVectors  | 0        | 0           | 416      | 0               |
| ATAX        | 4688     | 745         | 1728     | 0               |
| Backprop    | 0        | 0           | 0        | 0               |
| BICG        | 8704     | 8385        | 9952     | 2224            |
| Hotspot     | 6144     | 0           | 4416     | 0               |
| MVT         | 2912     | 0           | 2736     | 0               |
| NW          | 29952    | 3230        | 23776    | 772             |
| Pathfinder  | 0        | 0           | 160      | 0               |
| Srad-v2     | 5632     | 3942        | 5632     | 3667            |
| 2DCONV      | 0        | 0           | 0        | 0               |
| StreamTriad | 0        | 0           | 0        | 0               |

The best results are bolded
Table 2 Total number of pages thrashed using different HPE-based methods under 125% memory oversubscription

| Benchmark   | Demand.+HPE | Tree.+HPE |
|-------------|-------------|-----------|
| AddVectors  | 0           | 377381    |
| ATAX        | 745         | 498928    |
| Backprop    | 0           | 14282372  |
| BICG        | 8385        | 33797724  |
| Hotspot     | 0           | 97340     |
| MVT         | 0           | 0         |
| NW          | 3230        | 8812785   |
| Pathfinder  | 0           | 1878699   |
| Srad-v2     | 3942        | 44650411  |
| 2DCONV      | 0           | 566653    |
| StreamTriad | 0           | 3690578   |

Section 3.1. **UVMSmart** exploits both page migration and zero-copy to handle memory oversubscription, which improves the baseline performance to some extent. However, the excessive use of pinned memory is risky, and this may hurt the performance of the applications that use paged memory. **D.+HPE** achieves the second-best result besides with the upper bound. We believe that this is due to the demand load (no garbage prefetching) and the novel eviction policy (more information for eviction candidates’ selection). After all, we can see that there is still space to improve between the SOTA works and the upper bound.

Table 2 shows that **HPE** experiences dramatic performance loss when it cooperates with a data prefetcher to handle memory oversubscription (**Tree.** indicates the tree-based prefetcher). **HPE** relies on the per-page counter, which records the number of touched pages in each basic block to classify applications into regular and irregular access pattern types. This counter is substantially affected by data prefetching, which makes it unable to deliver correct classification. In this case, we can see that adaptability between the prefetching mechanism and the eviction policy is necessary, and demand load is beneficial for reducing the number of pages thrashed. However, demand load cannot handle the performance loss caused by on-demand migration (described in Section 2). This motivates the requirement for more accurate prefetching, which means prefetching with high accuracy and coverage. This raises the need for learning-based data prefetching mechanisms.

3.3 Limitation of Existing Learning-Based Methods

Machine learning has provided insights into various computer architectural problems, including branch prediction [14, 15], cache replacement [5], and data prefetching [6, 16–19]. Most of these works can be formulated as follows:

\[
P(Access_{t+1} | Access_1, Access_2, \cdots, Access_t) \quad (1)
\]

Equation (1) indicates that correlations between consecutive historical memory access information from time step 1 to \( t \) can be exploited to predict future memory access at time step \( t+1 \). By leveraging
Table 3  Unique page deltas in different program phases

| Benchmark | Phase0 | Phase1 | Phase2 |
|-----------|--------|--------|--------|
| AddVectors | 55     | 56     | 56     |
| ATAX      | 112    | 113    | 114    |
| Backprop  | 45     | 131    | 141    |
| BICG      | 17     | 18     | 37     |
| Hotspot   | 59     | 66     | 71     |
| MVT       | 6      | 12     | 12     |
| NW        | 479    | 830    | 1466   |
| Pathfinder | 98     | 102    | 103    |
| Srad-v2   | 49     | 145    | 170    |
| 2DCONV    | 155    | 155    | 155    |
| StreamTriad | 38    | 38     | 38     |

machine learning, existing works show superior accuracy and coverage in data prefetching compared to the rule-based methods described in Section 3.2. However, there are limitations to these designs. Previous works [6, 19] present online design by performing the training and prediction process in an alternate manner to accommodate the requirements of hardware prefetchers. Intuitively, this design makes sense since memory access patterns may vary in different program phases. The neural model needs to be retrained to capture the dynamic pattern. However, we find that the reality may not be the same as our intuition.

Figure 4 shows the results of page delta prediction on a single GPGPU workload using the online training method and the offline training method. In the online

Fig. 5  Page delta distribution in different program phases

(a) NW  (b) Srad-v2

(c) StreamTriad  (d) Hotspot
training experiments, we use the starting 10% instructions (approximately 50 million instructions in most cases) of each GPGPU workload to train the model, and we use the trained model to make predictions for the following 10% instructions. This train-predict loop continues until all the instructions are consumed. In the offline training experiments, we randomly select 50% of the total instructions as the training set to train the neural model, and then we use the trained model to make predictions for all the instructions in temporal order. Both the online experiments and the offline experiments use the same input-output pair (input data is a sequence of 10 consecutive historical memory accesses, input features include address, address delta, and PC; output is the following memory access of the input sequence, output class is address delta), the same neural model (transformer), and the same hyperparameters.

Figure 4 shows that there is an average 11.08% top-1 accuracy loss of page delta prediction using online training compared with using offline training. NW, Hotspot, and Srad-v2 are the three workloads that experience the largest accuracy loss (NW’s accuracy loss is 51.4%, Hotspot’s accuracy loss is 16.2%, Srad-v2’s accuracy loss is 11.4%). We believe that the analysis of these workloads can reveal the reason for this performance gap between online training and offline training.

Table 3 shows the number of unique page delta (the classification class used in the experiments described in Fig. 4) in different program phases. We can see that NW’s and Srad-v2’s page delta numbers grow significantly from the beginning to the end. The distribution of NW’s (Fig. 5(a)) and Srad-v2’s page delta (Fig. 5(b)) also vary in different phrases. We believe that these are the major reasons for the accuracy loss of NW’s and Srad-v2’s page delta prediction using online methods: In each training batch, the neural model is updated with new coming training data whose number of classes is continually growing. Ideally, the neural model should be able to learn new patterns while maintaining the ability to recognize previous patterns. However, machine learning or deep learning models always suffer from serious forgetting problems when they are continuously updated with new coming data. This is also termed the ‘catastrophic forgetting’ problem.

According to Table 3 and Fig. 5(c)-(d), Hotspot’s and StreamTriad’s page delta share a similar steady characteristic in both the quantity and the distribution. However, StreamTriad’s online prediction loss is smaller than that of Hotspot. We reference the deterministic finite automaton (DFA) described in [2] to re-label all the input data with their corresponding access pattern. According to DFA’s definition, there are 6 access patterns (described in Section 4.3) so we use the digits 0-5 to represent them. Figure 6(a)-(b) show the visualization of the access stream of StreamTriad and Hotspot workloads in different program

![Fig. 6](image-url) Memory access pattern visualization in different program phases. Each heat level is corresponding to a specific memory access pattern
phrases using these new labels. Compared to Hotspot, we can see that there is stronger temporal proximity of similar access patterns within StreamTriads’ visualization. We believe that such pattern proximity helps the neural model perform better in a near local range within the application’s global memory access stream. However, it is difficult for a single neural model to capture the knowledge in an online manner when the distribution of the memory access patterns is sparse. In this case, blindly increasing the training epoch of each batch may trigger the overtraining problem, which causes even worse prediction performance in the subsequent batch. We find that it is better to use separate models to make predictions for different memory access patterns. For Hotspot’s page delta prediction, offline training delivers the highest top-1 accuracy (85.6%). Online training using multiple models delivers the second-best performance (80.5%), and online training using a single model experiences the largest performance loss (69.4%).

4 Our Solution

We describe our intelligent framework for oversubscription management in CPU-GPU UVM in this section. We start by presenting a high-level overview of the framework. We then describe the key innovations in our design. First, to handle the potentially explosive growing number of classes in the new incoming data, our framework uses incremental learning. Second, a novel neural model is used to perform data prefetching prediction with thrashing information awareness. Third, our framework adopts a pattern-aware scheme, so that instead of relying on a single model to handle all the data, our framework uses separate models to learn the knowledge from the data of different access patterns. Fourth, a policy engine is designed to make decisions on prefetching and eviction decisions according to the neural model output. Fifth, we discuss the hardware complexity of our solution.

4.1 Overall Design and Workflow

As shown in Fig. 7, our framework (the gray blocks) is as follows: (1) Since some of the variables (thread block ID etc.) may not be available past GMMU, a memory access history register (MAHR) is added in the GMMU to collect data and feed them to the UVM runtime; (2) After receiving the extracted features (page address, page address delta, PC, thread block ID), there is a pattern classifier to classify the sequences into different access-pattern categories; (3) Upon the classification result, a specific predictor corresponding to the certain pattern is selected and be used to perform prediction; (4) The policy engine decides the next memory strategy according to the prediction results and delivers it to GMMU; (5) GMMU receives the policy and performs the corresponding operations (including data prefetching, data eviction, pinning, etc.).

4.2 Thrashing-Aware Incremental Learning-Based Page Predictor

Figure 8 shows our framework’s neural model’s architecture. To extract knowledge from both regular (data reuse, strides, etc.) and irregular (complex calculation with memory indirection, pointer chase, etc.) access patterns, the framework’s neural model exploits four input features: page address, page address delta, PC, and thread block ID (TB ID). More precisely, address and delta are used to capture the regular access pattern, while PC and TB ID are used to capture the irregular access pattern. Thus, the two pairs of input features are fed to two separate blocks (regular and irregular) for training after their individual embedding. Either block is an independent transformer [7]. The reason for using the transformer is described in Section 5.3. Finally, the outputs of these two blocks will be weighted by a learnable parameter respectively, and then they will be concatenated and be fed
into a linear layer, producing a probability distribution over the classification classes. We use page delta as the output class in this study. The dimension of the embedding layers needs to adapt for each application since the predictor cannot able to know the number of page deltas or PCs in advance. Similarly, the linear layer’s dimension is also unpredictable. Either the regular block or the irregular block is an encoder-only transformer similar to BERT [21]. The input dimension of the regular block is 10 $\times$ 128 (10 is the length of the input sequence, 64+64=128 is the sum of the page address and the page delta after embedding, and 64 is the virtual address space of the x86-64 platform). The input dimension of the irregular block is 10 $\times$ 32 (10 is the length of the input sequence, and 16+16=32 is the sum of the PC and the thread block ID after embedding, we empirically assume that both the largest PC and the largest thread block ID will not exceed $2^{16} = 65536$). The batch size is 32. After that, a positional embedding layer is applied. We use the original position encoding scheme [7], which is based on a family of sinusoidal functions. The output of the positional encoding layer will be input to a stack of transformer encoders (we use a stack size of 2), which perform the multiheaded self-attention evaluation among the tokens within the input sequence and generate output encodings. Finally, the outputs of the regular and the irregular blocks are concatenated and processed through linear transformation and softmax. This finishes one forward pass of our predictor. The input dimension of the linear layer is the same as the concatenation of the encodings (10 $\times$ 128 + 10 $\times$ 32 = 1600). The output dimension of the linear layer needs to adapt for each benchmark application.

A separate model is trained for each access pattern (this is further discussed in Section 4.3). We randomly select 5 benchmarks (ATAX, Backprop, BICG, Hotspot, NW) and run them using different input sets compared to the experiments described in Section 5. We use 50% of each of these benchmarks’ UVM traces to build a corpus, and we use this corpus as the training dataset to train the 6 generic predictors until their accuracy reaches a reasonable range ($\geq 0.85$). We use these pre-trained models to make predictions for 11 memory-intensive GPGPU benchmarks’ 100% UVM requests (Section 5). These predictors do not need to be retrained until the inference loss exceeds a predefined threshold.

As described in Section 3.3, we introduce incremental learning to handle the potential catastrophic forgetting problem caused by a continually growing number of classes over time. To avoid the knowledge acquired from the previous data being over-ridden by the new coming data, a regulation term (LUCIR [22]) is added to the model’s loss function to consolidate previous knowledge when learning new data. It is worth noting that our framework can also adapt to other more advanced regulation-based methods.

Since our final goal is to reduce oversubscription overhead, the page predictor should consider thrashing information. We introduce a thrashing term into the page predictor’s loss function, which is computed as follows:

$$L_{Thra}(x) = \sum_{i=1}^{|E| \cup |T|} y_i \log(p_i)$$

(2)
\(E\) is the set of pages that have been evicted from the GPU device memory to the CPU memory, and \(T\) is the set of pages that have already been thrashed. \(y\) is the one-hot ground-truth label and \(p\) is the corresponding class probability obtained by softmax. Mathematically, \(L_{Thra}(x)\) is the additive inverse of the standard cross-entropy (CE) loss. \(L_{Thra}(x)\) encourages the model to deliver page predictions beyond the set of evicted pages and thrashed pages, which aims to reduce the probability of pages being repetitively thrashed.

In total, the aggregated loss function \(L\) is as follows:

\[
L = \frac{1}{|\mathcal{N}|} \sum_{x \in \mathcal{N}} (L_{CE}(x)+\lambda L^G_{dis}(x)) + \frac{\mu}{|\mathcal{S}|} \sum_{x \in \mathcal{S}} L_{Thra}(x) \tag{3}
\]

\(\mathcal{N}\) is a training batch, which is composed of a set of pages corresponding to the newly arrived memory accesses. \(\mathcal{S}\) is a subset of \(\mathcal{N}\), which is defined as \(|\mathcal{S}| = |\mathcal{N}| \cap (|\mathcal{E}| \cup |\mathcal{T}|). L_{CE}(x)\) is the standard CE loss. \(L^G_{dis}(x)\) is the regulation term introduced by LUCIR [22], which is used to encourage the orientation of features extracted by a current neural model to be similar to those extracted by the previous model. \(\lambda\) and \(\mu\) are both loss weights. \(\lambda\) is used to adjust the degree of need to preserve the previous knowledge according to the number of new classes introduced in each training batch. \(\mu\) is used to adjust the degree of need to migrate the pages in the historical page thrashing and page eviction record according to the current memory access pattern. Intuitively, there are three goals that \(L\) aims to achieve: First, \(L\) uses \(L_{CE}(x)\) to perform multiclass classification learning upon the historical memory accesses. Second, \(L\) uses \(L^G_{dis}(x)\) to improve model training under the multiclass incremental setting. Third, \(L\) uses \(L_{Thra}(x)\) to mitigate the probability of predicting pages that are potentially being thrashed.

### 4.3 Pattern-aware Prediction Scheme

It is costly to apply a separate model for each application in both time and memory. On the other hand, it is also not amenable to expect there exists one universal model worked for all the applications. This is because different GPU workloads have different access patterns. We reference DFA [2] to analyze the selected benchmarks in this study. DFA’s mechanism is briefly described as follows. Based on the group of far-faults from GPU, unified memory runtime (UVM backend) determines the 64KB basic blocks as the migration candidates. These basic blocks are communicated to the I/O root complex to schedule DMA transfers. After segregating these transfers at kernel boundaries based on their scheduling timestamp, DFA scans the corresponding basic block addresses and determines whether they show linearity/randomness of migration. In addition, basic block addresses are compared to determine any re-referencing across the kernel boundaries. According to our exploration, some of the observed patterns of the selected benchmarks are shown in Table 4.

\(A\) and \(B\) denote access to a virtual memory page. \(k\) denotes the number of accesses. \(N, N_A\) and \(N_B\) denotes the frequency that data is re-referenced (reused). \(\epsilon\) denotes the probability of touching that page. The streaming pattern represents a temporal sequence of accesses repeating \(N\) times. When \(k\) exceeds the GPU device memory size, the streaming pattern turns into a thrashing pattern where part of the data will be repeatedly referenced and transferred between the CPU and the GPU memory. The random pattern represents a temporal sequence of accesses with different reference frequencies. Each part or most of the sequence can be iterated over different iterations.

| Pattern          | Representation                                                                 |
|------------------|-------------------------------------------------------------------------------|
| Streaming/Thrashing | \((A_1, A_2, \ldots, A_k)^N, N \geq 0\)                                     |
| Random           | \((A_1^{N_1}\epsilon_1, A_2^{N_2}\epsilon_2, \ldots, A_k^{N_k}\epsilon_k)^N, N \geq 0\) |
|                  | \(N_1, N_2, \ldots, N_k \geq 1, 1 \geq \epsilon_1, \epsilon_2, \ldots, \epsilon_k \geq 0\) |
| Mixed            | \(((A_1^{N_1}\epsilon_1, A_2^{N_2}\epsilon_2, \ldots, A_p^{N_p}\epsilon_p)^N + (B_1, B_2, \ldots, B_q)^N, N \geq 0\) |
|                  | \(N_1, N_2, \ldots, N_p \geq 1, 1 \geq \epsilon_1, \epsilon_2, \ldots, \epsilon_p \geq 0, N_A, N_B \geq 0\) |
**Table 5** Improved DFA’s detectable patterns and the corresponding metrics

| Pattern               | Metrics                                                                 |
|-----------------------|-------------------------------------------------------------------------|
| [Regular]Streaming     | γdfa = Str., Cr ≥ C1, ldfa = Str., flagR = 0                           |
| [Regular]Random        | γdfa = Ran., Cr ≥ C1, ldfa = Ran., flagR = 0                           |
| [Regular]Mixed_S       | γdfa = Mix., Cr ≥ C1, ldfa = Str., flagR = 0                           |
| [Regular]Mixed_R       | γdfa = Mix., Cr ≥ C1, ldfa = Ran., flagR = 0                           |
| [Regular]Streaming_reuse| γdfa = Str., Cr ≥ C1, ldfa = Str., flagR = 1                           |
| [Regular]Random_reuse  | γdfa = Ran., Cr ≥ C1, ldfa = Ran., flagR = 1                           |
| [Regular]Mixed_S_reuse | γdfa = Mix., Cr ≥ C1, ldfa = Str., flagR = 1                           |
| [Regular]Mixed_R_reuse | γdfa = Mix., Cr ≥ C1, ldfa = Ran., flagR = 1                           |
| [Irregular]Streaming   | γdfa = Str., Cr < C1, ldfa = Str., flagR = 0                           |
| [Irregular]Random      | γdfa = Ran., Cr < C1, ldfa = Ran., flagR = 0                           |
| [Irregular]Mixed_S     | γdfa = Mix., Cr < C1, ldfa = Str., flagR = 0                           |
| [Irregular]Mixed_R     | γdfa = Mix., Cr < C1, ldfa = Ran., flagR = 0                           |
| [Irregular]Streaming_reuse| γdfa = Str., Cr < C1, ldfa = Str., flagR = 1                           |
| [Irregular]Random_reuse| γdfa = Ran., Cr < C1, ldfa = Ran., flagR = 1                           |
| [Irregular]Mixed_S_reuse| γdfa = Mix., Cr < C1, ldfa = Str., flagR = 1                           |
| [Irregular]Mixed_R_reuse| γdfa = Mix., Cr < C1, ldfa = Ran., flagR = 1                           |

\( \gamma_dfa \) denotes the output of the DFA automaton. \( l_dfa \) denotes the input of the DFA automaton. Str. denotes streaming. Ran. denotes random. \( C_R \) denotes the number of frequently accessed pages. \( C_I \) denotes the number of rarely accessed pages. \( \text{flagR} \) denotes the binary value representing whether there are reused pages within the input sequence.

respectively. This pattern can be further categorized into the part repetitive pattern and the most repetitive pattern [3]. The mixed pattern represents a composition or nesting of the streaming and the random pattern. We improve DFA by further distinguishing the memory access patterns with the consideration of the pages’ access frequencies. For instance, the distribution of the streaming accesses and the random accesses is not uniform within the traces belonging to the mixed pattern among different benchmark kernels. Instead of applying a single model for all the mixed pattern data, using separate models to learn the streaming and the random part respectively may deliver better prediction performance. In addition, as for the traces belonging to the same pattern, it is difficult for a single model to learn the heavily accessed pages (hot) and the rarely accessed ones (cold) together. It is appropriate to separate the learning of the hot and the cold pages to deliver correct predictions. Furthermore, pages sharing among multiple kernels show distinct patterns compared to the ones without data reuse. Table 5 shows the patterns that the improved DFA can detect and the metrics that are applied to detect the related pattern. We use the improved DFA as the pattern classifier in this study.

To demonstrate our pattern detection method, we use the two kernels (ATAK1, ATAK2) of the ATAX benchmark as an example. Figure 9(a) shows that ATAX1 exhibits sparse memory access among different data structures. Although ATAX1’s 3 PCs’ access patterns seem to be similar, stronger linearity is found residing in PC 336 and PC 464 compared to PC 200 when we zoom in on ATAX1’s accesses. Figure 9(b) shows that ATAX2 exhibits sequential, dense, and repetitive page access over the allocation boundary. This shows the diversity of the access patterns within one application and this verifies the necessity of using separate models to capture each pattern’s knowledge correctly. Figure 9(c) and (d) show that ATAX1’s access frequency (maximum: 600) is significantly lower than ATAX2’s (maximum: >600). In this case, if a single model is applied to make predictions for ATAX1 and ATAX2, the prediction results will unavoidably bias to ATAX2 due to the imbalance of the input samples, and this leads to incorrect predictions for ATAX1 in the end. We defined the pages whose access frequency is lower than 10% of the maximum access frequency of the entire application as the rarely accessed pages, and the rest are categorized as the frequently accessed pages. We use the number of these two kinds of pages as the metric to categorize an access sequence as regular or irregular (Table 5). Figure 9(c) and (d) also show that all the data structures accessed by ATAX1 and ATAX2 are read-only. Figure 9(a) and (b) show that pages around 78700 are reused between
Fig. 9 Visualizing page access patterns of the ATAX1 and the ATAX2 kernels.
Fig. 10 Percent stacked bar chart of 11 GPGPU benchmarks’ memory access patterns. Srad indicates Srad-v2. ATAX1 and ATAX2 indicate the first and the second kernel of the ATAX benchmark. This goes the same for the Backprop, BICG, NW, and Srad-v2 benchmarks.

ATAX1 and ATAX2. For these pages, a binary flag (flagR) is set and the relative access sequences are selected and learned distinguishably compared to the others belonging to the same access pattern. This is because the reused pages are more likely to trigger serious thrashing compared to the pages which will never be touched once they are accessed. Overall, according to Table 5, our pattern detection method characterizes ATAX1’s accesses as [Irregular] Streaming_reuse, [Irregular] Random_reuse, [Irregular] Mixed_S_reuse, and [Irregular] Mixed_R_reuse. And ATAX2’s accesses are characterized as [Regular] Streaming_reuse, [Regular] Mixed_S_reuse, and [Regular] Mixed_R_reuse. Figure 10 shows the percentage of different access patterns within the 11 GPGPU benchmarks’ memory traces.

Fig. 11 Overview of intelligent framework’s policy engine

4.4 Prediction-Based Memory Strategy

When the GPU device memory fills to capacity, page eviction is needed. As described in Fig. 11, the eviction candidates are selected according to the page predictor’s prediction results. To alleviate the probability of instant thrashing, we leverage the page set chain proposed by Yu et al. [3], which classifies the accessed pages into three partitions (new, middle, old) according to intervals (a specified number of page faults). We use this page set chain and update it with both the demand loads and the prefetches. The search for eviction candidates migrates from the old partition to the new partition depending on whether the certain partition is empty or not. In addition, we use a prediction frequency table to select the eviction candidate.
within the same partition. This frequency table keeps counters for pages occurring in several near intervals’ predictions, which indicates the importance of these pages in the near future memory accesses of the workload. When a certain partition is chosen and the search begins, the eviction candidate will be selected from the pages with the lowest prediction frequency. For pages that never appear in the prediction results, their frequencies are set to $-1$. This frequency table will be flushed periodically to maintain an accurate representation of memory accesses in different program phases. We use the same interval length (64) as HPE [3], and we empirically flush the page frequency table every 3 intervals.

To deal with the long latency caused by on-demand paging, prefetching is needed. Data prefetching candidates are generated from the prediction results in each interval. An identical prediction frequency table as the eviction phase is used in data prefetching (Fig. 11). This table is used to control the aggressiveness of prefetching when the oversubscription level is too high. Prefetching candidates will be selected from the pages with the highest prediction frequency. To mitigate the TLB shootdown overhead, the basic blocks relative to the latest touched page and the page with the highest prediction frequency will also be selected to be prefetched.

4.5 Hardware Complexity

Table 6 shows the memory consumption of the pattern-aware prediction scheme. As described in Section 4.2, some of the components’ dimensions need to adapt for each application, so Table 6 discusses the memory usage corresponding to each benchmark respectively. These memory consumption statistics are collected using an MIT-licensed library [23]. Equation (4) shows the calculation of the total memory footprint.

$$Total = (Params. \times 2 + Acti.) \times Patterns$$  \hspace{1cm} (4)

($Params. \times 2 + Acti.$) indicates that both the weights of the current and the previous models need to be stored for the calculation of the LUCIR term, but only the current activation is needed to update the model’s weights. This sum is multiplied by Patterns because each access pattern needs an individual model to make predictions. The model corresponding to a specific pattern is generic to all the benchmarks. We exploit quantization and pruning to compress the memory consumption. We find that clamping the weights and the forward/backward pass activation to [-16,+16] will not harm the predictor’s performance. This indicates that the memory footprint can be mitigated compared to the one using the float32 value (5 bits is enough to represent all the values in our predictor). Overall, the largest storage cost of the predictor is 1.74 MB, which is significantly smaller than that of a state-of-the-art deep-learning-empowered data prefetcher [6, 24]. In addition, a new transformer engine using FP16 precision has been built upon the latest NVIDIA Hopper Tensor Core [25]. We believe that the hardware complexity of our solution can be further improved in the latest GPU architecture.

The prediction frequency table is designed as a 16-way set associate cache. Each entry corresponds

| Benchmark | Parameters (MB) | Activation (MB) | Patterns | Total (MB) |
|-----------|----------------|----------------|----------|------------|
| AddVectors | 0.06           | 0.24           | 4        | 1.44       |
| ATAX      | 0.02           | 0.18           | 7        | 1.54       |
| Backprop  | 0.15           | 0.28           | 7        | 4.06       |
| BICG      | 0.16           | 0.22           | 5        | 2.7        |
| Hotspot   | 0.08           | 0.25           | 4        | 1.64       |
| MVT       | 0.09           | 0.19           | 4        | 1.48       |
| NW        | 0.06           | 0.25           | 9        | 3.33       |
| Pathfinder | 0.10           | 0.23           | 6        | 2.58       |
| Srad-v2   | 0.07           | 0.26           | 8        | 3.20       |
| 2DCONV    | 0.06           | 0.25           | 5        | 1.85       |
| StreamTriad | 0.06       | 0.23           | 4        | 1.4        |

两会 Springer
to a basic block (described in Section 2.2), which means the tag is 48 bits (system width 64 bits, page address width 12 bits, basic block address width 16 bits). The data field of each entry is used to store the counter of the predicted pages within each basic block. Since we flush this table every 3 intervals, a 6-bit counter for each page is adequate. The length of this frequency table is 1024. Thus, the total storage overhead of the prediction frequency table is 18 KB ((6*16+48)/8*1024=18 KB), which is relatively small compared to the pattern-aware prediction scheme.

5 Evaluation

First, we compare prediction performance running with the previous training methods and our solution. Second, we compare prediction overhead sensitivity, IPC, reduction in thrashing, and scalability running with our solution and a SOTA framework (UVMSmart [2]) that supports delayed page migration, zero-copy, and tree-based page prefetching.

5.1 Evaluation Methodology

To compare with different learning-based methods, we implement different model-based predictors in PyTorch. We trained these models with each GPGPU workload’s data using the online and offline training methods (described in Section 3.3) and our solution. To compare with different memory strategies under oversubscription, we use a GPGPU-Sim extension implemented by Ganguly et al. [2] in our experiments. This extension provides functional and timing simulation support for UVM. This extension also provides a set of regular and irregular GPU applications from the Rodinia, Lonestar, and Polybench benchmark suites. These benchmarks are modified to use CUDA UVM APIs (cudaMallocManaged, cudaMemcpyAsync, and cudaMemcpySynchronize). Table 7 shows the primary configuration of the simulator, and the configuration associated with the UVMSmart runtime is the same as in [2].

5.2 Normalized Execution Time

We compare the execution time of the benchmark kernels running with the proposed intelligent framework against the execution time of kernels running with the SOTA design (UVMSmart [2]). Note that some of the benchmark kernels (ATAX, NW, 2DCONV) running with UVMSmart crash (>10X) under heavy oversubscription (150%). This is because UVMSmart’s eviction policy fails to generate eviction candidates under memory oversubscription while most of the device memory is not evictable (being processed by the LDST, or being staged or scheduled for write-back, etc.), and this keeps the GPU warps stalling for the requested pages and crashes the kernel in the end. By prefetching and evicting less amount of pages with high precision and coverage, our solution reduces the probability of encountering the dilemma where both the device memory and the evictable pages are drained, and our solution can handle this problem by identifying proper eviction candidates to prevent the GPU kernel from crashing. Without considering the crashed kernels, Fig. 12 shows that the proposed framework provides an average (geometric mean) 28.81% and 198.09% execution time reduction under 125% and 150% memory oversubscription respectively.

5.3 Prediction Performance

Figure 13 shows the comparison results of using different predictors to deliver page delta prediction. We can see that the transformer-based predictor delivers the best prediction performance compared to the other traditional models (CNN, LSTM, MLP). MemMAP [26] is a novel meta-learning-based approach for data...
prefetching. This approach is built upon the compressed LSTM model and it achieves better performance than the traditional LSTM. Zhang et al. further proposed TransMAP [27] exploiting the transformer model and it achieves similar performance compared to the results of the transformer-based predictor. This further verifies the effectiveness of the transformer in data prefetching. Overall, we select the transformer as the regular and irregular block neural model in our framework.

Offline training (profiling-based training) exploits future memory access information for training. We consider offline training performance as the upper bound of the page prediction. Our solution alleviates the performance gap between the online and offline training (Fig. 14) by exploiting incremental learning (Section 4.2) and a pattern-aware prediction scheme (Section 4.3). Compared to the online training method, our solution achieves an average 6.45% top-1 accuracy improvement (41.2% at most).

Figure 15 shows that our solution (w. term) achieves an average 7.31% page thrashing reduction while causing a minimal accuracy degradation (0.42%) compared to the one using only LUCIR loss (w/o. term). The loss weight $\mu$ is adaptively adjusted ranging from (0, 1] according to the applications’ different memory access patterns. For instance, a larger thrashing term value benefits the streaming access pattern where

**Fig. 13** Performance using different page predictors

---

**Fig. 12** Normalized execution time of 11 GPGPU benchmarks under 125% and 150% memory oversubscription. All the results are normalized by the execution time running with UVMSmart under 125% memory oversubscription.
Fig. 14 Normalized top-1 accuracy using online training, offline training, and our solution.

Pages are hardly re-referenced once they are accessed. However, a smaller thrashing term value benefits the most repetitive access pattern where pages are referenced multiple times with different frequencies.

5.4 Prediction Overhead Sensitivity Tests

As described in Section 4.2, we randomly select 5 benchmarks’ data (ATAX, Backprop, BICG, Hotspot, NW) to build a corpus, and we train our predictor relative to each pattern on this corpus until their accuracies reach a reasonable range (≥0.85). We use the pre-trained models to make predictions for each benchmark. According to our statistics among 11 benchmarks, this training method introduces a microsecond-level inference overhead for each batch’s prediction. Shi et al. [6] claim that their model can make predictions every 18000 nanoseconds. NVIDIA leveraged the TensorRT8.0 SDK [28] to slash BERT-large’s (345 million parameters) prediction overhead.
Fig. 16  Normalized IPC results using our solution with different levels of overhead under 125% oversubscription.

![Normalized IPC results using our solution with different levels of overhead under 125% oversubscription.](image)

Fig. 17  Normalized IPC results using variant memory strategies under 125% and 150% oversubscription. All the results are normalized by the IPC running with UVMSmart under 125% memory oversubscription.

![Normalized IPC results using variant memory strategies under 125% and 150% oversubscription.](image)

to 1.2 $ms$. We believe that the prediction overhead can be improved by more advanced hardware/software technologies, more fancy equipment, and more sophisticated programming skills. However, these are beyond the scope of this study. We conduct a prediction overhead sensitivity test of our predictor.

We vary the prediction overhead from 1, 10, 20, 50, and 100 microseconds per batch of prediction (batch size is 32). Since the GPU core frequency is configured as 1481 MHz in the simulator, these overhead candidates roughly correspond to 1500, 15000, 30000, 75000, and 150000 cycle-per-batch-of-prediction in each simulation. We consider UVMSmart as a SOTA design. Figure 16 shows that the average normalized IPC under different levels of overheads are 1.52X, 1.32X, 1.17X, 0.91X, and 0.71x. These results show that the learning-based methods are sensitive to the overhead.

A transformer engine was built upon the latest NVIDIA Hopper Tensor Core [25]. This engine uses FP8 and FP16 precisions to reduce memory storage and increase performance. Since our transformer-based predictor uses a smaller precision than FP8 (Section 4.3), both the hardware and prediction overhead of adopting our solution is ideally negligible for the latest GPU architecture. Thus, we use 1 microsecond (1500 cycles) as the prediction overhead for batched UVM requests in the following evaluation. This is sharply distinct from the previous works [5, 6] that consider zero prediction overhead while exploiting deep learning models to boost the application’s IPC performance. The training overhead of the pre-trained model is not considered in the sim-
ulation, and we assume that it can be achieved offline in practice.

5.5 IPC

Figure 17 shows that our solution achieves 3% to 140% performance improvement for 11 benchmarks under 125% memory oversubscription. When the oversubscription level grows to 150%, some of the benchmarks (ATAX, NW, 2DCONV) using UVMSmart crashed due to serious page thrashing. Our solution prevents these benchmarks from crashing through accurate prefetching and pre-eviction, and it achieves 131% to 328% performance improvement for the other benchmarks. Overall, our solution achieves average IPC improvement (geometric mean) of 1.52X and 3.66X under 125% and 150% memory oversubscription compared to the SOTA design.

5.6 Reduction in Thrashing

Table 8 shows that our solution (Ours.) achieves the least number of pages thrashed compared to other methods (Base., T.+H., UVMS.) involving both data prefetching and eviction. The cooperation between the prefetching mechanism and the eviction policy is improved by applying a thrashing-aware predictor and a shared data structure (Section 4.4). There is still space to improve between our solution and the methods (D.+H., D.+B.) involving demand loads. This result indicates that our solution cannot deliver perfect prediction, and there are still useless prefetches compared to demand loads. Without considering T.+H. (which is malfunctioning and is described in Section 3.2) and the demand-load-based methods (D.+H., D.+B.), our solution achieves an average 64.4% reduction in pages thrashed compared to the baseline under 125% oversubscription, while the SOTA works (UVMS.) achieves an average 17.3% reduction compared to the baseline. Page reduction under 150% memory oversubscription is not discussed in this section because some of the benchmarks using UVMSmart crashed.

5.7 Scalability

Modern GPUs allow multiple kernels or applications to share a single device concurrently [29]. This feature raises the challenge of handling UVM oversubscription under a higher level of kernel concurrency compared with running a single GPU workload. We test the scalability of our solution by running multiple concurrent workloads belonging to different access patterns (streaming, regular, mixed, random). Table 9 shows that our solution achieves an average 10.2% top-1 accuracy improvement (30.2% at

| Bench. | with Prefetching | without Prefetching |
|--------|------------------|---------------------|
|        | Base. | T.+HP | UVMS. | Our sol. | D.+HP | D.+Belady. |
| AddVec. | 0 | 377381 | 416 | 60 | 0 | 0 |
| ATAX | 4688 | 498928 | 1728 | 936 | 745 | 0 |
| Backprop | 0 | 14282372 | 0 | 1 | 0 | 0 |
| BICG | 8704 | 33797724 | 9952 | 8398 | 8385 | 2224 |
| Hotspot | 6144 | 97340 | 4416 | 31 | 0 | 0 |
| MVT | 2912 | 0 | 2736 | 0 | 0 | 0 |
| NW | 29952 | 8812785 | 23776 | 6651 | 3230 | 772 |
| Pathfinder | 0 | 1878699 | 160 | 7 | 0 | 0 |
| Srad-v2 | 5632 | 44650411 | 5632 | 4209 | 3942 | 3667 |
| 2DCONV | 0 | 566653 | 0 | 0 | 0 | 0 |
| S.T. | 0 | 3690578 | 0 | 0 | 0 | 0 |

Base. indicates Baseline, T. indicates Tree, UVMS. indicates UVMSmart, sol. indicates solution, D. indicates Demand, AddVec. indicates AddVectors, S.T. indicates StreamTriad. The best results are bolded.
Table 9: Top-1 accuracy of page delta prediction on multiple workloads using online training and our solution

| Type       | Benchmark          | Online training | Our solution |
|------------|--------------------|-----------------|--------------|
| Str.+Str.  | StreamTriad+2DCONV | 0.908           | 0.874        |
| Str.+Reg.  | StreamTriad+Srad-v2| 0.772           | 0.800        |
| Reg.+Str.  | Hotspot+2DCONV     | 0.744           | 0.804        |
| Reg.+Reg.  | Hotspot+Srad-v2    | 0.683           | 0.76         |
| Mix.+Str.  | NW+2DCONV          | 0.560           | 0.849        |
| Mix.+Reg.  | NW+Srad-v2         | 0.478           | 0.780        |
| Ran.+Str.  | ATAX+2DCONV        | 0.837           | 0.867        |
| Ran.+Reg.  | ATAX+Srad-v2       | 0.716           | 0.780        |

Str. indicates Streaming, Reg. indicates Regular, Mix. indicates Mixed, Ran. indicates Random.

5.8 Evaluation with NVIDIA cuDNN Library

To verify the effectiveness of our proposed framework on the modern popular machine learning workloads, we implement our design on the latest GPGPU-Sim 4.0 [30] and extend our evaluation to two samples provided by the NVIDIA cuDNN library [31]. The first sample application is a program of convolution which performs common machine learning operations such as forward, backward data, and backward filter convolutions. The second application models LeNet [32] trained with the MNIST dataset. These samples are modified to use CUDA UVM APIs as described in Section 5.1. Overall, these sample applications exhibit more fine-grained data sharing between the CPU and the GPU compared to the traditional GPU benchmarks, and they are more similar to real-world workloads using nowadays popular machine learning frameworks such as TensorFlow and Pytorch.

Figure 18 shows the memory access patterns of the two kernels invoking the cuDNN APIs. Note that both of these sample applications consist of multiple kernels. For simplicity, Fig. 18 only shows the memory access patterns of 1 kernel of each application.

We compare the IPC of the sample kernels running with our proposed framework against the execution time of the kernels running with UVMSmart (Fig. 19). As for the cuDNN-conv workload, we can see that our solution only achieves slight performance improvement (4.86% on average, geometric mean) compared to UVMSmart in all the experiments. This is because the cuDNN-conv workload only experiences a small amount of page thrashing under 125% and 150% memory oversubscription (Table 10). Besides, UVMSmart also achieves a high device memory page hit rate by using its tree-based prefetcher (87.28%). In this case, there is not much opportunity for performance improvement by leveraging more accurate prefetching or pre-eviction. On the other hand, our solution achieves higher performance improvement (11.98% on average, geometric mean) on the cuDNN-mnist workload. cuDNN-mnist experiences a higher level of page thrashing compared to cuDNN-conv under all levels of memory oversubscription. In addition, UVMSmart’s tree-based prefetcher only achieves a 33.11% device page hit rate. Our solution improves the page hit rate to 40.02% while reducing 53.49% of the device-to-host traffic. This means that our solution help reduces the unnecessary pages brought to the device memory, which saves both the competitive PCI-e bandwidth and the limited device memory under heavy memory oversubscription. This helps the application to endure a more serious oversubscription circumstance. Overall, these experiments show that our solution can be adapted to improve the modern deep-learning workloads performance under GPU memory oversubscription.

6 Related Works

This paper is the first to propose an incremental-learning-based approach for CPU-GPU UVM oversubscription management. We now discuss related works in UVM using other methods, and studies...
(a) cuDNNconv accesses distinguished by PC.  (b) cuDNNmnist accesses distinguished by PC.

(c) cuDNNconv access distribution detailing type of accesses.
(d) cuDNNmnist access distribution detailing type of accesses.

(e) cuDNNconv accesses distinguished by detected patterns.
(f) cuDNNmnist accesses distinguished by detected patterns.

Fig. 18 Visualizing page access patterns of the cuDNNconv and the cuDNNmnist kernels
that apply artificial intelligence to other parts of the microarchitecture.

6.1 CPU-GPU UVM Oversubscription Studies

UVM support in modern discrete CPU-GPU systems [33, 34] has been studied widely. Agarwal et al. [10] proposed aggressive first-touch migration and prefetching neighboring pages. Zheng et al. [12] studied different user-directed and user-agnostic prefetchers to overlap data migration and kernel execution. Ganguly et al. [9] uncovered the mechanism of the tree-based prefetcher implemented in the NVIDIA GPU driver. Pratheek et al. [35] proposed walk stealing to reduce interference in page walks from concurrent tenants while also ensuring high walker utilization. Oversubscription overhead has become a first-order overhead for data-intensive GPGPU applications. VAST [36] partitions data-parallel workloads according to available GPU physical memory. GPUswap [37] relocates data from the GPU to the host RAM and makes it accessible to the device under oversubscription. Agarwal et al. [38] proposed a compiler-based profiling mechanism to make programmers aware of the specific memory access pattern. Ganguly et al. [9] proposed a tree-based page replacement policy inspired by the tree-based prefetcher in CUDA driver to deal with memory oversubscription. Chen et al. [39] proposed an application-transparent framework for reducing memory oversubscription overheads in GPUs. Kim et al. [40] proposed a GPU runtime software and hardware solution that enables efficient demanding paging for GPUs. Ganguly et al. proposed a programmer-agnostic framework [41] and an application-aware adaptive framework [2] to deal with memory oversubscription overhead stemming from page thrashing in irregular, data-intensive GPU applications. Yu et al. [3] proposed a hierarchical page eviction policy that addresses LRU’s inability to handle thrashing access patterns while retaining LRU’s advantages for LRU-friendly patterns. Yu et al. [42] proposed a coordinated page prefetch and eviction design to manage oversubscription for GPUs with unified memory. NVIDIA developers[43] explored different designs to improve GPU memory oversubscription performance on various platforms from x86 to P9, and V100 and A100 GPUs.

### Table 10

Total number of pages thrashed using our solution and UVMSmart under 125% and 150% memory oversubscription

| Benchmark    | UVMSmart 125% | UVMSmart 150% | OurSol 125% | OurSol 150% |
|--------------|---------------|---------------|-------------|-------------|
| cuDNNconv    | 5             | 128           | 0           | 32          |
| cuDNNmnist   | 160           | 1520          | 15          | 619         |
6.2 Artificial Intelligence in Computer Architecture

Seznec et al. [44] have made the case for considering the TAGE predictor in real hardware processors through new directions. Hashemi et al. [4] apply the RNN model to the analysis of memory access patterns, which demonstrates higher precision and recall than table-based approaches. Peled et al. [16] proposed the context-based prefetcher, which employs the contextual bandits model of reinforcement learning. Bhatia et al. [17] introduced perceptron-based prefetch filtering which acts as an independent check on the quality of predictions made by the underlying prefetch engine. Shi et al. [5] applied deep learning to solve the cache replacement problem. Doudali et al. [45] presented a page scheduler with machine intelligence for applications that execute over hybrid memory systems.

Peled et al. [18] use a fully-connected feed-forward network instead, and they formulate prefetching as a regression problem to train their neural network. Shi et al. [6] propose a hierarchical model of data prefetching that accommodates both delta patterns and addresses correlation. Bera et al. [19] propose a customizable prefetching framework that formulates prefetching as a reinforcement learning problem.

7 Conclusion

In this paper, we propose a transformer-based intelligent framework for oversubscription management in CPU-GPU unified virtual memory. As far as we know, it is the first work using deep learning for oversubscription management in CPU-GPU unified memory. We first provide an in-depth analysis of the current rule-based methods for oversubscription management. We identify the necessity of applying machine intelligence for accurate data prefetching and eviction to improve the rule-based methods, and we identify sources of performance overhead for the current learning-based method are the explosive growing number of classes and the nonpattern awareness. Then, we design a framework to solve the problems. We enable the pattern awareness of the page predictor by classifying the input data into different categories according to their access patterns, and we train them with a separate neural model. We introduce incremental learning to help the predictor handle the continually growing number of classes. We use a policy engine to deliver prefetching or eviction decisions according to the prediction results. Finally, the evaluation results show that our solution outperforms the SOTA designs. Furthermore, our solution can be adapted to modern GPU architecture and provide performance improvement for real-world workloads.

Acknowledgments This work was supported in part by the Industrial Internet Innovation and Development Action Plan Project (No. TC210A02K) and the National Natural Science Foundation of China (No.61802022 and No.61802027).

Author Contributions Xinjian Long: Conceptualization of this study, Data curation, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Visualization, Writing-original draft. Xiangyang Gong: Methodology, Project administration, Resources, Supervision, Writing-review & editing. Bo Zhang: Methodology, Funding acquisition, Project administration, Resources, Supervision, Writing-review & editing. Huiyang Zhou: Conceptualization, Methodology, Funding acquisition, Project administration, Resources, Supervision, Writing-review & editing.

Funding This work was supported in part by the Industrial Internet Innovation and Development Action Plan Project (No. TC210A02K) and the National Natural Science Foundation of China (No.61802022 and No.61802027).

Availability of data and materials Data sharing is not applicable.

Compliance with Ethical Standards

Competing interests We declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Consent for Publication We confirm that we understand journal Journal of Grid Computing is a transformative journal. When research is accepted for publication, there is a choice to publish using either immediate gold open access or the traditional publishing route.

Ethics approval and consent to participate The results/data/figures in this manuscript have not been published elsewhere, nor are they under consideration by another publisher. We have read the Springer journal policies on author responsibilities and submit this manuscript in accordance with those policies. All of the material is owned by the authors and/or no permissions are required.
References

1. Sakharnykh, N.: Everything you need to know about unified memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf (2018)

2. Ganguly, D., Melhem, R., Yang, J.: An adaptive framework for oversubscription management in cpu-gpu unified memory. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1212–1217. IEEE (2021)

3. Yu, Q., Childers, B., Huang, L., Qian, C., Wang, Z.: Hpe: Hierarchical page eviction policy for unified memory in gpus. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 39(10), 2461–2474 (2019)

4. Hashemi, M., Swersky, K., Smith, J., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., Ranganathan, P.: Learning memory access patterns. In: International Conference on Machine Learning. pp. 1919–1928. PMLR (2018)

5. Shi, Z., Huang, X., Lin, C.: A hierarchical neural model of data prefetching. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 413–425 (2019)

6. Shi, Z., Jain, A., Swersky, K., Hashemi, M., Ranganathan, P., Lin, C.: A hierarchical neural model of data prefetching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 861–873 (2021)

7. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. Advances in Neural Information Processing Systems, vol. 30 (2017)

8. Appelhans, D., Auerbach, G., Averill, D., Black, R., Brown, A., Buono, D., Cash, R., Chen, D., Deindl, M., Duffy, D., et al: Functionality and performance of nvlink with ibm power9 processors. Ibm Journal of Research and Development, vol. 62(4-5) (2018)

9. Ganguly, D., Zhang, Z., Yang, J., Melhem, R.: Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 224–239 (2019)

10. Agarwal, N., Nellans, D., O’Connor, M., Keckler, S.W., Wenisch, T.F.: Unlocking bandwidth for gpus in cc-numa systems. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 354–365. IEEE (2015)

11. Sakharnykh, N.: Unified memory on pascal and volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf (2017)

12. Zheng, T., Nellans, D., Zulfiqar, A., Stephenson, M., Keckler, S.W.: Towards high performance pagged memory for gpus. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 345–357. IEEE (2016)

13. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems journal 5(2), 78–101 (1966)

14. Tarsa, S.J., Lin, C.-K., Keskin, G., Chinya, G., Wang, H.: Improving branch prediction by modeling global history with convolutional neural networks. arXiv:1906.09889 (2019)

15. Zangeneh, S., Pruett, S., Lym, S., Patt, Y.N.: Branchnet: a convolutional neural network to predict hard-to-predict branches. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 118–130. IEEE (2020)

16. Peled, L., Mannon, S., Weiser, U., Etsion, Y.: Semantic locality and context-based prefetching using reinforcement learning. In: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), pp. 285–297. IEEE (2015)

17. Bhatia, E., Chacon, G., Pugsley, S., Teran, E., Gratz, P.V., Jiménez, D.A.: Perceptron-based prefetch filtering. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), pp. 1–13. IEEE (2019)

18. Peled, L., Weiser, U., Etsion, Y.: A neural network prefetcher for arbitrary memory access patterns. ACM Transactions on Architecture and Code Optimization (TACO) 16(4), 1–27 (2019)

19. Bera, R., Kanellopoulos, K., Nori, A., Shahroodi, T., Subramoney, S., Mutlu, O.: Pythia: A customizable hardware prefetching framework using online reinforcement learning. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 1121–1137 (2021)

20. Parisi, G.I., Kemker, R., Part, J.L., Kanan, C., Wermer, S.: Continual lifelong learning with neural networks: a review. Neural Netw. 113, 54–71 (2019)

21. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805 (2018)

22. Hou, S., Pan, X., Loy, C.C., Wang, Z., Lin, D.: Learning a unified classifier incrementally via rebalancing. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 831–839 (2019)

23. TylerYep: torchinfo. https://github.com/TylerYep/torchinfo (2021)

24. Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., Yang, M.: Estimating gpu memory consumption of deep learning models. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1342–1352 (2020)

25. NVIDIA: NVIDIA hopper architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/ (2022)

26. Srivastava, A., Wang, T.-Y., Zhang, P., Rose, C.A.F.D., Kannan, R., Prasanna, V.K.: Memmap: compact and generalizable meta-Istm models for memory access prediction. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 57–68. Springer (2020)

27. Zhang, P., Srivastava, A., Nori, A.V., Kannan, R., Prasanna, V.K.: Transformap: transformer for memory access prediction. arXiv:2205.14778 (2022)

28. Rodge, J.: NVIDIA announces tensorRT 8 Slashing BERT-Large inference down to 1 millisecond. https://developer.nvidia.com/blog/nvidia-announces-tensorrt-8-slașing-bert-large-inference-down-to-1-millisecond/ (2021)
29. NVIDIA: MULTI-PROCESS SERVICE https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf (2021)
30. Lew, J., Shah, D.A., Pati, S., Cattell, S., Zhang, M., Sandhuatla, A., Ng, C., Goli, N., Sinclair, M.D., Rogers, T.G.: Analyzing machine learning workloads using a detailed gpu simulator. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 151–152. IEEE (2019)
31. NVIDIA: NVIDIA cuDNN. https://developer.nvidia.com/cudnn (2022)
32. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE 86(11), 2278–2324 (1998)
33. AMD: Radeons Next-generation Vega architecture. https://radeon.com/downloads/vega-whitepaper-11.6.17.pdf (2017)
34. NVIDIA: Pascal architecture Whitepaper. https://images.nvidia.cn/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (2019)
35. Pratheek, B., Jawalkar, N., Basu, A.: Improving gpu multi-tenancy with page walk stealing. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 626–639. IEEE (2021)
36. Lee, J., Samadi, M., Mahlke, S.: Vast: the illusion of a large memory space for gpus. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 443–454. IEEE (2014)
37. Kehne, J., Metter, J., Bellosa, F.: Gpuswap: enabling oversubscription of gpu memory through transparent swapping. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 65–77 (2015)
38. Agarwal, N., Nellans, D., Stephenson, M., O’Connor, M., Keckler, S.W.: Page placement strategies for gpus within heterogeneous memory systems. In: Proceedings of the Twentieth International Conference on Architectural and Operating Systems, pp. 607–618 (2015)
39. Li, C., Ausavarungnirun, R., Rossbach, C.J., Zhang, Y., Mutlu, O., Guo, Y., Yang, J.: A framework for memory oversubscription management in graphics processing units. In: Proceedings of the Twenty-fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 49–63 (2019)
40. Kim, H., Sim, J., Gera, P., Hadidi, R., Kim, H.: 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, pp. 1357–1370 (2020)
41. Ganguly, D., Zhang, Z., Yang, J., Melhem, R.: Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 451–461. IEEE (2020)
42. Yu, Q., Childers, B., Huang, L., Qian, C., Guo, H., Wang, Z.: Coordinated page prefetch and eviction for memory oversubscription management in gpus. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 472–482. IEEE (2020)
43. NVIDIA: Improving GPU Memory Oversubscription Performance https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/ (2021)
44. Seznec, A.: A new case for the tage branch predictor. In: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture, pp. 117–127 (2011)
45. Doudali, T.D., Blagodurov, S., Vishnu, A., Gurumurthi, S., Gavrilovska, A.: Kleio: A hybrid memory page scheduler with machine intelligence. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, pp. 37–48 (2019)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.
Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.