Coordinated Page Prefetch and Eviction for Memory Oversubscription Management in GPUs

Qi Yu†, Bruce Childers‡, Libo Huang†(✉), Cheng Qian†, Hui Guo†, Zhiying Wang†
†*School of Computer, National University of Defense Technology and* ‡*University of Pittsburgh*
{*yuqi13, libohuang, qiancheng, huiguo, zywang*}*@nudt.edu.cn, childers@pitt.edu*

***Abstract*—The adoption of unified memory and demand paging has simplified programming and eased memory management in discrete GPUs. However, long-latency page faults cause significant performance overhead. While several software-based mechanisms have been proposed to address this issue, they suffer from inefficiency when page prefetching and pre-eviction are combined. For example, a state-of-the-art page replacement policy, hierarchical page eviction (HPE), is inefficient when prefetching is enabled. Furthermore, the prefetcher semantics-aware pre-evicting policy, which pre-evicts continuous pages in bulk the way they were brought in by the prefetcher, may cause thrashing for some irregular applications.**

In this paper, *coordinated page prefetch and eviction (CPPE)* is proposed to manage memory oversubscription in GPUs with unified memory. CPPE incorporates a modified page eviction policy, MHPE, and an access pattern-aware prefetcher in a fine-grained manner: MHPE is aware of prefetch semantics and the prefetcher prefetches pages according to access patterns in eviction candidates selected by MHPE. Simulation results show that, when the GPU memory is 75% and 50% oversubscribed, CPPE achieves an average speedup of 1.56x and 1.64x (up to 10.97x) over the state-of-the-art baseline, which combines a sequential-local prefetcher and LRU pre-eviction policy. CPPE also outperforms other approaches, including Random/reserved LRU with the sequential-local prefetcher, and simply disabling prefetching under memory oversubscription.

***Index Terms*—GPUs, Unified Memory, Eviction Policy, Prefetching, Access Pattern**

I. Introduction

In the past decade, the performance of Graphics Processing Units (GPUs) has improved due to transistor scaling [1] [2] and architecture innovation. GPUs are widely used in many application domains, such as graph analytics, machine learning, and multi-media [3]. Two types of GPUs are common in the marketplace: on-die GPUs and discrete GPUs. Despite significant investment in on-die GPUs, discrete PCIe attached GPUs are more common due to their flexibility and high performance, as well as power and thermal constraints of their on-die counterparts [4].

Recent support for *unified memory* [5] and *demand paging* [6] in GPUs has eased programming and increased programmer productivity by eliminating explicit copies and memory management. In unified memory, CPU and GPU memory share a single, unified virtual address space. The GPU's software runtime system pages memory in and out of the GPU on demand [7]. These capabilities permit transparently overlapping data transfer and kernel execution. Additionally, a GPU can process datasets that exceed the GPU's memory capacity with memory *oversubscription*. Unfortunately, there is no free lunch. Because GPUs do not support context switches to operating system service routines, GPU page table management and page transfers are managed by the GPU's runtime system on the host CPU [8]. When a page fault occurs, several PCIe round trips and interaction with the host CPU are required, which impose significant overhead, possibly more than 20 microseconds [9] [10].

To mitigate performance degradation caused by page faults, prior research proposed *locality prefetch* [9] and *pre-eviction* [11]. By prefetching (pre-evicting) a chunk of continuous pages, the cost (including page table updates and CPU-GPU communication) for handling a single page fault (eviction) is amortized over multiple page faults (evictions); as a consequence, the overhead is reduced. Another possibility is to optimize the eviction (replacement) policy under memory oversubscription to reduce page faults. In particular, it is well known that LRU has difficulty with thrashing access patterns [12] [13]. To address this issue, Q. Yu *et al.* proposed *hierarchical page eviction (HPE)* [14] [15]. For thrashing access patterns, HPE selects eviction candidates according to MRU rather than LRU, which helps alleviate thrashing. D. Ganguly *et al.* proposed *reserved LRU* [16], which avoids selecting the top portion (percentage) of the LRU page list as eviction candidates. To permit page eviction with prefetching, D. Ganguly *et al.* further proposed a "pre-eviction policy" that is aware of prefetch semantics [16]. This policy pre-evicts contiguous pages in the order in which they were brought in by the prefetcher.

HPE and reserved LRU suffer from inefficiency in certain situations. HPE is designed for GPUs that do not have prefetch support. This policy relies on per page set counter, which records the number of touched pages in a page set, to classify applications into regular and irregular access pattern types. When prefetching is enabled, the per page set counter records the number of *prefetched* pages rather than *touched* pages. This change in page set counter's functionality makes it unable to correctly classify the pattern type. Reserved LRU has limited performance improvement for applications with thrashing access patterns. It even harms the performance of some irregular applications. D. Ganguly *et al.*'s approach, which incorporates prefetch semantics in the pre-eviction policy, chooses to naïvely prefetch a whole chunk of pages when GPU memory is exhausted. This coarse-grained solution incurs performance slowdown for some irregular applications.

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00056

In this paper, we explore the opportunity for a fine-grained cooperation between page prefetch and eviction to manage oversubscription of GPUs with unified memory. For this purpose, we propose *coordinated page prefetch and eviction (CPPE)*. To permit effective page eviction with page prefetching, CPPE uses a modified version of HPE (termed *MHPE*) to handle thrashing access patterns. To facilitate page prefetching with page eviction, CPPE introduces an access pattern-aware prefetcher, which prefetches pages according to the access pattern in eviction candidates selected by MHPE. Compared to a state-of-the-art software baseline, which combines a sequential-local prefetcher and LRU pre-eviction policy (proposed in [16]), CPPE achieves an average of 1.56x and 1.64x speedup (up to 10.97x) when the GPU memory is 75% and 50% oversubscribed, respectively.

This paper makes the following contributions:

- An analysis of existing software mechanisms is presented, including HPE, reserved LRU, and D. Ganguly *et al.*'s approach [16]. We demonstrate their inefficiency and motivate the need to develop a more effective approach when prefetching is enabled under memory oversubscription.
- CPPE is proposed to manage memory oversubscription for GPUs. CPPE integrates a new eviction policy, Modified Hierarchical Page Eviction (MHPE), with a new access pattern-aware page prefetcher in a fine-grained manager.
- We evaluate CPPE to determine some key parameters for the scheme. We also compare it against several prior approaches and demonstrate that CPPE outperforms them, in some cases, significantly.

II. Background

Section II-A describes unified GPU memory, including the components for address translation and how address translation is performed. Section II-B presents prefetch strategies for unified memory. Section II-C introduces basic principle of hierarchical page eviction policy.

A. Unified Memory

Support for *memory virtualization* requires address translation. Although NVIDIA, AMD, and Intel have not revealed details of the memory hierarchy for their GPU architectures, it is generally accepted that current GPUs use TLB-based address translation [17]. J. Power *et al.* [18] and B. Pichai *et al.* [19] were among the first to explore such designs. J. Power *et al.* showed that (1) per-SM (Streaming Multiprocessor) post-coalesced TLBs; (2) a highly-threaded page table walker; and, (3) a shared page walk cache are essential for efficient address translation. Subsequent research demonstrated the effectiveness of a shared L2 TLB [17]. Fig. 1 depicts a baseline GPU architecture that supports address translation (shaded components). This baseline, which we adopt, is the same as prior work [11] [20]. In this design, each SM has a private L1 TLB. These per-SM L1 TLBs are backed by a L2 TLB shared by all SMs. The highly-threaded page table walker supports multiple concurrent walks.

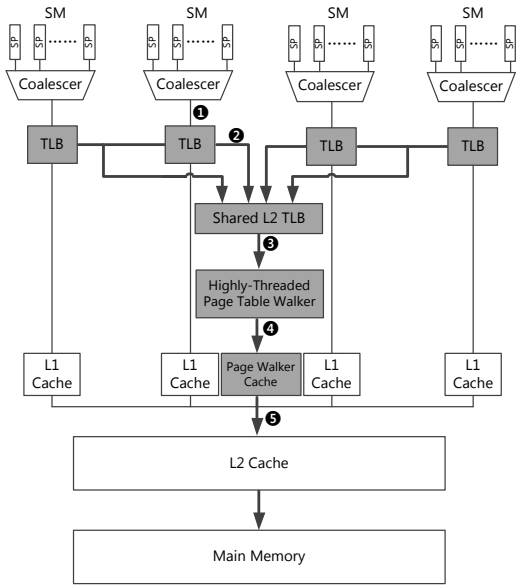

Fig. 1: Address translation in a GPU

Memory requests first access the L1 TLB to check for a virtual-to-physical address translation (❶ in Fig. 1). On an L1 TLB miss, the request is sent to the L2 TLB (❷). If the request also misses in the L2 TLB, the page table walker begins a walk (❸). The walk starts with a probe to the shared page walk cache (❹). A walk that misses in the page walk cache goes to the L2 cache, and if necessary, to main memory (❺). If the page table walker does not find the address mapping in the page table, a page fault is taken. Because current GPUs do not support precise exceptions [21], page migration is offloaded to the GPU's software runtime system, which executes on the host CPU. This end-to-end process has significant overhead due to frequent interaction with the host CPU.

B. Prefetch Strategy

Prefetching pages can reduce compulsory misses and alleviate the influence of long-latency page faults. In GPUs with unified memory, T. Zheng *et al.* proposed a locality prefetcher which identifies prefetch candidates from the next sequential 128 virtual pages following the faulting page [9]. Using microbenchmarks, D. Ganguly *et al.* discovered that NVIDIA CUDA driver uses a tree-based neighborhood prefetcher [16]. These prefetch strategies mitigate performance loss due to expensive page faults for regular applications [9] [11] [16]. However, prior work also argued that continuing to prefetch pages when the GPU memory is exhausted may hurt performance of some irregular applications.

C. Hierarchical Page Eviction Policy

HPE dynamically maintains a chunk chain (we use "chunk" rather than "page set" [15] to be consistent with prior work [16] and NVIDIA's terminology [22]). A *chunk* is a group of virtual pages with continuous addresses. An application's

execution is partitioned into multiple *intervals*, where each interval is equal to a specified number of page faults. The structure of chunk chain is shown in Fig. 2. The chunk chain is divided into three partitions according to chunk recency. *Old partition* contains chunks that were previously referenced but have not been referenced in the last and current intervals. *Middle partition* contains chunks that were referenced in the last interval. *New partition* contains chunks that are referenced in the current interval. Each chunk has an entry in the chain, and the entry has a counter field to record the number of touches to the chunk. When GPU memory fills to capacity, HPE first selects a chunk, and then the virtual pages in the chunk are selected in address order and the corresponding physical pages are evicted to the CPU memory.

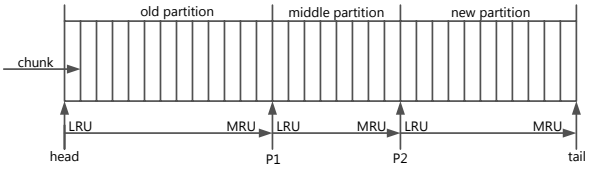

Fig. 2: Structure of chunk chain

HPE uses the chunk counter to classify applications into three categories: regular, irregular#1, and irregular#2. For regular applications, HPE uses a strategy that considers both chunk recency and frequency, which is called MRU-C (MRU counter). MRU-C searches from the **MRU** position of the **old** partition until a qualified chunk is found. For irregular#1 and irregular#2 applications, HPE starts with LRU, which selects a chunk from the LRU position of the **old** partition. To deal with misclassification and variance in access behavior at runtime, HPE dynamically adjusts the eviction strategy. For regular applications, HPE remains with MRU-C but adjusts the search start point to find eviction candidates. For irregular#1 applications, HPE remains with LRU. For irregular#2 application, HPE switches between MRU-C and LRU by comparing the number of intervals a strategy lasts.

III. Motivation

Existing software-based techniques to manage memory oversubscription, including HPE, reserved LRU, and D. Ganguly *et al.*'s approach all suffer in certain situations. This motivates the need for a mechanism to coordinate page prefecth and eviction in a fine-grained manner.

Inefficiency 1: *HPE is inefficient with page prefetch.* HPE aims to address LRU's disadvantages without considering prefetching pages (i.e., prefetching is disabled) [14]. This policy relies on chunk counters for efficient classification, which track the number of touched pages in a chunk. When page prefetching is enabled with HPE, most pages in a chunk are *prefetched* rather than previously *touched*. For example, when a page fault occurs, the faulted page and the remaining pages in a chunk are migrated to GPU memory with prefetching. In this case, the counter is set to the chunk size (due to migrating all pages in the chunk). However, only one page is actually touched and the remaining pages are prefetched. In HPE, the counters are used to differentiate regular and irregular applications to select an appropriate eviction strategy, once the GPU memory fills to capacity. Because the counters are polluted by page prefetches, they have lost the information needed to classify an application into regular or irregular type.

One way to address this issue is to record reference information on the GPU side and transfer this information to the GPU driver via the PCIe bus, just like [15] does. However, this method causes extra interruption of the GPU driver, incurring additional overhead. Yet, it is difficult for HPE to determine touched pages without reference information. Therefore, in the context of GPUs that support page prefetch, how to make HPE work efficiently is a challenge.

Inefficiency 2: *Reserved LRU has limited performance gains for thrashing access patterns and may degrade performance of some irregular applications.* Reserving the top percentage of LRU page list reduces thrashing for thrashing access patterns [16]; however, the performance improvement is usually limited. In certain situations, this method may even hurt performance of some irregular applications.

To show the limitations of reserved LRU, we compared it against LRU and Random eviction policies. We coupled all three policies with a locality prefetcher, which prefetches a chunk (16 pages) each time, same as prefetching the 64KB basic block [16]. Pre-eviction is also enabled, which evicts a chunk each time (the same granularity with prefetch). For simplicity, we report results only for 50% memory oversubscription. The experimental setup and a description of benchmarks can be found in Section V. Fig. 3 shows the results. In the figure, LRU-20% means the top 20% of LRU chunk chain is reserved. The first four applications have thrashing access patterns, and the last two are irregular applications.

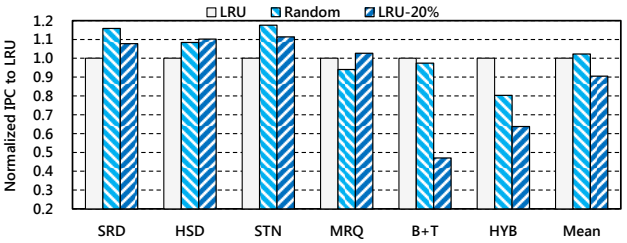

Fig. 3: Comparison of LRU with Random and reserved LRU.

According to the figure, reserved LRU achieves limited speedup for applications with thrashing access patterns (at most 11%). In some cases, the gain is actually lower than Random (*SRD* and *STN*). For *B+T* and *HYB*, reserved LRU suffers from significant performance degradation (up to 53%). On average, reserved LRU performs worse than LRU and Random for these applications. In addition, the performance improvement is related to the reservation percentage, which is difficult to determine without *a priori* profiling of an application.

Inefficiency 3: *Prefetching naïvely once memory is oversubscribed incurs severe performance degradation for some irregular applications.* Prefetching pages can be effective to mitigate overhead caused by long-latency page faults. However, for some irregular applications, when the GPU memory fills to capacity, continuing prefetching naïvely as [16] does may incur thrashing, and even cause a program to crash. To quantify the thrashing degree, we compared two cases: (1) prefetching pages for the entirety of execution; and (2) turning off prefetching when the GPU memory fills to capacity. Both cases use locality prefetching and LRU pre-eviction policy. We use the number of page evictions as the metric of interest. Result is normalized to the second case. We show only the applications with a normalized result greater than 1.2. For the remaining applications, the difference between the two cases is within 20%.

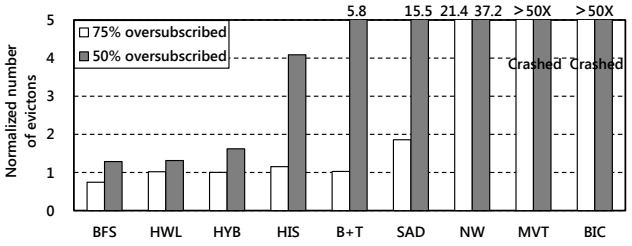

Fig. 4: Sensitivity to prefetching once memory is full.

Fig. 4 shows the amount of thrashing due to prefetching pages once memory capacity is oversubscribed. Generally, continuing to prefetch pages leads to more page evictions. *SAD* and *NW* had an order of magnitude more evictions with prefetching. *MVT* and *BIC* crashed during execution due to severe thrashing, which corroborates prior observations [11]. Although past research [11] [16] proposed to turn off prefetching once memory is exhausted to alleviate thrashing, we found that this is not a one-size-fits-all solution. Indeed, it may cause performance slowdown for regular applications and certain irregular applications that experience less thrashing (see Section VI-B for detailed results).

In summary, prior work has two major issues. First, the proposed eviction policy suffers from inefficiency in certain cases, e.g., HPE achieves notable performance improvement over LRU but does not support prefetching, and reserved LRU is compatible with prefetching but achieves limited performance gain over LRU. Second, prefetching naïvely or simply turning off prefetching under memory oversubscription is not sufficient. Therefore, to efficiently manage oversubscription for GPUs with unified memory, fine-grained cooperation between prefetching strategy and eviction policy is necessary. The eviction policy needs to address LRU's disadvantages while supporting page prefetching. The prefetch strategy needs to prefetch pages more smartly.

IV. Coordinated Page Prefetch and Eviction

We propose a fine-grained solution, Coordinated Page Prefetch and Eviction (CPPE), to manage memory oversubscription for GPUs with unified memory. CPPE addresses the inefficiency of prior work as follows. First, CPPE adopts a modified HPE (MHPE) policy as the eviction policy. MHPE retains both HPE's advantage for thrashing access patterns and flexibility to adjust the eviction strategy. MHPE also enhances HPE to be compatible with page prefetching. Second, CPPE augments the locality prefetcher to be aware of access patterns in evicted chunks. With these features, CPPE performs well for regular applications, especially for applications with thrashing access patterns. It also mitigates thrashing caused by the coarse-grained mechanism [16] for irregular applications.

A. Overview of Approach

CPPE includes MHPE and an access pattern-aware locality prefetcher. MHPE relies on the number of untouched pages in evicted chunks rather than chunk counters to classify applications into regular and irregular types. On this basis, MHPE modifies the mechanisms to select an eviction strategy, adjust the search point, and switch the eviction strategy at runtime. CPPE's access pattern-aware prefetcher relies on the touch pattern in evicted chunks to decide which pages to prefetch, rather than prefetching a whole chunk [16] or simply turning off prefetching [11] under memory oversubscription. These two techniques are incorporated in a fine-grained manner: MHPE selects a chunk (prefetched by the locality prefetcher) to evict and the prefetcher prefetches pages according to the touch pattern in evicted chunks (selected by MHPE).

B. MHPE: Modifying HPE to Support Page Prefetch

HPE addresses LRU's poor performance for thrashing access patterns by using MRU-C strategy. For irregular access patterns, HPE dynamically switches between LRU and MRU-C strategy to minimize bad eviction decisions. This approach has been shown to work well to manage page eviction in GPUs [14] [15]. For this reason, we adopt HPE as the starting point for CPPE's eviction policy, MHPE (see Algorithm 1). HPE relies on chunk counters to differentiate regular and irregular access pattern types, and selects an appropriate eviction strategy (between LRU and MRU-C) for each type.

MHPE uses the same base configuration as HPE: the chunk size is 16 and the interval length is 64. When a chunk is prefetched to GPU memory, a bit vector is initialized for the chunk. All bits in the vector are set to 1 and the metadata of this chunk (including the bit vector) is inserted at the tail of the chunk chain. As explained earlier, the counter-based classification used by HPE is not effective with page prefetching. Instead, MHPE does not record access counts to chunks. Inspired by the observation that "chunks of regular applications usually need fewer intervals to be fully populated" [14], MHPE uses the number of untouched pages in evicted chunks to classify applications into regular and irregular types. This eliminates the count information used by HPE. Without the counters, MRU-C devolves into MRU, which beneficially reduces search overhead to find eviction candidates. The untouched pages can be determined from the bit vector, which records touches to individual pages in a chunk. Chunks of

Algorithm 1 MHPE

```
1: Definition:
2: U1 ← total untouch level in one interval
3: U2 ← total untouch level in the first four intervals
4: W ← number of wrong evictions in one interval
5: Initialization:
6: strategy ← MRU
7: Calculate initial forward distance
8: while program is not over do
9:     Calculate U1, U2, and W
10:    if strategy = MRU then
11:        if U1 >= T1 or U2 >= T2 then
12:            strategy ← LRU
13:        else
14:            if forward distance <= T3 then
15:                forward distance ← compare(U1, W)
16:            end if
17:        end if
18:    end if
19: end while
```

regular applications tend to have fewer untouched pages than irregular applications when these chunks are evicted. To ease explanation, the "lifetime" of a chunk is the number of chunks prefetched between the time when a chunk is prefetched and the time when that chunk is evicted.

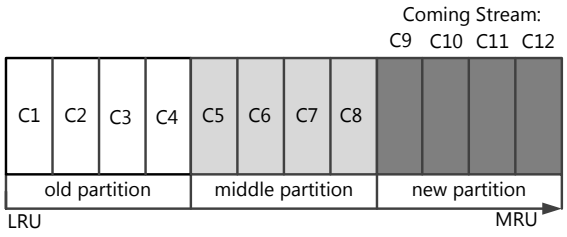

Fig. 5: Example of lifetime.

Fig. 5 shows an example of lifetime. Suppose the GPU memory becomes full when eight chunks are prefetched. These chunks (C1-C8) are inserted into the chunk chain according to prefetched order. When a page fault occurs, the next chunk, C9, should be prefetched into GPU memory. To make room for C9, a chunk in the old partition is selected. C1 is evicted under LRU with a lifetime of 8. Alternatively, C4 is evicted under MRU with a lifetime of 5. The chunks at the MRU position have shorter lifetime than chunks at the LRU position. As a result, they tend to have more untouched pages when they are evicted. A common way to extend the lifetime of chunks at the MRU position is to skip several chunks from the MRU position. For instance, if two chunks are skipped, C2 will be evicted (with a lifetime of 7) under MRU. In this case, the lifetime of C4 will be increased to 6 if it is evicted to make room for C10. We define two more terms for explanation: "untouch level" is the number of untouched pages in an evicted chunk; and, "forward distance" is the number of chunks skipped from the MRU position.

Selecting the Eviction Strategy. Without counter information for chunk accesses, classifying applications and selecting an appropriate eviction strategy (LRU/MRU) is difficult. Instead, when the GPU memory fills to capacity, MHPE starts with the MRU policy (see Algorithm 1, line 6), which it may dynamically switch to LRU. We choose MRU as the starting policy for two reasons. First, MRU outperforms LRU for thrashing access patterns. Second, chunks at the MRU position tend to have more untouched pages than chunks at the LRU position, which makes it easier to differentiate between regular and irregular applications. However, in practice, we found that the chunk at the exact MRU position of some regular applications (e.g., *HSD*, *SRD*, etc.) has a similar untouch level as some irregular applications.

Thus, to differentiate between regular and irregular applications, MHPE forwards the search point by several chunks from the MRU position. To find an appropriate "forward distance", we did a sensitivity test that evaluated distances of $1 \sim 10$. Results show that the untouch level of some regular applications drops significantly when the forward distance is larger than or equal to 2. When the distance is over 8, the untouch level of some irregular applications drops significantly, making it difficult to classify regular/irregular applications. On this basis, the appropriate forward distance range is $2 \sim 8$. Generally, different applications prefer a different forward distance. Consequently, MHPE dynamically selects the forward distance for each application. It uses the chunk chain length to determine an application's forward distance. When GPU memory is full, MHPE divides the chunk chain length by 100, and compares the result with the range boundaries (2 and 8). If the result is in the range, the initial forward distance is set to the result. Otherwise, the initial forward distance is set to the closest boundary value. The initial distance is calculated at line 7 of Algorithm 1.

Switching the Eviction Strategy. To respond to runtime variation, MHPE can dynamically switch from MRU to LRU, if the untouch level reaches two thresholds, as shown in Algorithm 1. To determine the thresholds, we studied the untouch level of several applications. Typically, an application's untouch level drops with execution; i.e., the untouch level of later intervals is lower than the first few intervals during execution. This is particularly true for some irregular applications, which have similar untouch level (in later intervals) with regular applications. This makes it difficult to differentiate between regular and irregular applications. Therefore, we record the untouch level in the first few intervals for analysis. Using this information, we classified the benchmarks into three categories:

- *High-Untouch*: Most pages (up to 15) in an evicted chunk are not touched. In this case, the untouch level is high and LRU outperforms MRU.
- *Medium-Untouch*: The number of untouched pages in

an evicted chunk is moderate, with around half pages receiving no touches. Although the untouch level is lower than the first category, LRU still outperforms MRU.
- *Low-Untouch*: Few pages in an evicted chunk are untouched. For this category, MRU performs better than LRU for trashing patterns and performs similarly for streaming patterns.

Based on this analysis, two conditions are introduced to switch from MRU to LRU:

1) If the total untouch level in an interval goes above a threshold, *T1*, switch from MRU to LRU. This condition handles High-Untouch applications.
2) If the total untouch level in the first four intervals goes above a threshold, *T2*, switch from MRU to LRU. This condition handles Medium-Untouch applications.

For simplicity, we put the two conditions to switch eviction strategy in a single line (11) in Algorithm 1. *U2* is compared to *T2* only once (at the fourth interval). In other intervals, MHPE only compares *U1* to *T1*. Note, as LRU outperforms MRU for high-untouch and medium-untouch applications, once the eviction strategy is switched to LRU, it is not switched back to MRU. This is different from HPE, in which the eviction strategy may switch between LRU and MRU.

Adjusting Forward Distance. Although regular applications have a relatively lower untouch level, MRU may incur thrashing with a fixed forward distance. Through analysis, we identified two sources of thrashing. One source is from untouched pages, which cause additional page faults after the corresponding chunk (to which the untouched page belongs) is evicted. Another source is the difference in access time between SMs. For example, SM#1 might access a page at $t1$, and SM#2 might access the same page at $t2$. If the corresponding chunk is evicted at $t3$ ($t1 < t3 < t2$), a page fault will occur on SM#2.

MHPE considers both sources of thrashing to adjust the forward distance. For the first source, MHPE uses the untouch level in each interval to guide adjustment. For the second source, MHPE accounts for the number of wrong evictions (similar to HPE [14] [15]) in each interval. A buffer is allocated for an application to record recently evicted chunks. When a page fault occurs, the buffer is searched for the corresponding chunk. On a hit, the number of wrong evictions is increased. MHPE determines the buffer length according to an application's chunk chain length, rather than using a fixed interval length [14] [15]. The minimum size of the buffer is 8 to store evicted chunks in the last two intervals. When the GPU memory is full, MHPE first divides the chunk chain length by 64 and multiples the result by 8. This method ensures the minimum length is 8 and a buffer with the same length is allocated for applications with a similar memory footprint. Wrongly evicted chunks are inserted at the head of the chunk chain (HPE inserts them at the tail). By keeping the wrongly evicted chunks in the LRU position and selecting eviction candidates from the MRU position, further thrashing caused by these chunks can be avoided.

As four chunks are prefetched in one interval, the number of wrong evictions ranges $0 \sim 4$. Correspondingly, MHPE divides the total untouch level (0∼T1-1) in an interval into five values. When an interval is over, MHPE calculates the untouch value and compares it against the number of wrong evictions. A larger value rather than sum (to avoid over adjustment) is selected and added to current forward distance. In addition, we find that some regular applications keep adjusting forward distance during execution, which in turn, hurts performance. To address this issue, a limit is set to forward distance: once the forward distance reaches a threshold (T3), the forward distance will not be further increased. Algorithm 1 shows this on line 14-15. The actual values used for T1, T2, and T3 are determined in Section VI-B.

C. Access Pattern-Aware Prefetch

Although chunks at the LRU position have a longer lifetime, we found that some chunks have high untouch level even if the eviction strategy is switched to LRU. We observed two reasons for this phenomenon. First, some chunks need many intervals to be fully populated, which is beyond LRU's ability to cover. Second, the specific pattern within a chunk may preclude prefetching the whole chunk. For example, only a portion of pages with a fixed stride (e.g., stride of 2 in *NW* and stride of 4 in *MVT*) are touched during a period of time. If the whole chunk is prefetched on a page fault, the pages that do not match the stride are not touched. Untouched pages compete for limited space with other pages, even hot pages, which can incur thrashing. In addition, transferring untouched pages over the PCIe bus wastes bandwidth and energy.

To address this issue, CPPE enhances the page prefetcher to be aware of the touch pattern in a chunk. A "pattern buffer" is used to record the touch pattern (touched pages) of evicted chunks. When a page fault occurs, the buffer is searched for the chunk corresponding to the faulted address. On a hit and the faulted page matches the touch pattern, only the touched pages in the chunk are prefetched. Otherwise, the whole chunk is prefetched. To reduce buffer length, only chunks that have an untouch level larger than or equal to 8 (i.e., a half of a chunk) are recorded. Chunks without a fixed pattern are removed from the buffer. We propose two schemes to delete chunks from the buffer. The first one deletes a chunk whenever a faulted page does not match the touch pattern. The second one deletes a chunk if a faulted page does not match the touch pattern during only the first search.

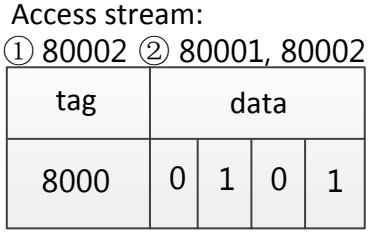

Fig. 6: Example of the two deletion schemes.

Fig. 6 shows an example of the difference between the two schemes. For simplicity, suppose a chunk has four continuous virtual pages. The tag is the chunk address and the data is a bit vector. A bit value of "0" means the page corresponding to that bit position is not touched (pages 80000 and 80002) and "1" indicates the page has been touched (pages 80001 and 80003). In the first access stream, page 80002 does not match the touch pattern; as a result, the whole chunk is prefetched and this chunk is deleted from the buffer under both schemes. The second access stream has two faulted pages that map to this chunk. Page 80001 matches the pattern; therefore, page 80001 and page 80003 are prefetched. The subsequent faulted page 80002 does not match the pattern; in this case, the whole chunk, except pages 80001 and 80003 (they have been prefetched), are prefetched. With the first scheme, the chunk is deleted from the buffer. With the second scheme, however, the chunk is left in the buffer because the first search (80001) is a pattern match. In Section VI-B, we compare the performance of the two schemes.

V. Experimental Methodology

GPGPU-Sim 3.2.2 [23] was enhanced with TLBs and a GPU Memory Management Unit (GMMU) [9] [17] [20] to support a unified memory. Similar to prior work [11] [17] [20], a two-level TLB design is used: each SM has a private L1 TLB, and all SMs share an L2 TLB. A shared page table walker is modeled that supports 64 concurrent page table walks and traverses a 4-level page table. An optimistic 20µs page fault latency [9] [11] is used. The replayable far-fault mechanism [9] was implemented to enable each SM to continue execution in the presence of page faults. A default OS page size of 4KB was adopted, similar to other research [9] [16] [18] [19] [24]. This page size is used in current GPUs [25]. The system configuration is shown in Table I.

TABLE I: Configuration of simulated system

GPU Cores	28 SMs, 1.4GHz
Private L1 cache	48KB, 6-way associative, LRU
Private L1 TLB	128-entry per SM, single port, 1-cycle latency, LRU, support hit under miss
Shared L2 cache	3MB total, 256KB/DRAM channel, 16-way associative, LRU
Shared L2 TLB	512-entry, 16-associative, LRU, 10-cycle latency, 2 ports
Page Table Walker	supporting 64 concurrent walks, traversing 4-level page table
Page Walk Cache	16-way 8KB, 10-cycle latency
DRAM	GDDR5, 12-channel, FR-FCFS scheduler, 528GB/s aggregate
CPU-GPU interconnect	16GB/s, 20 µs page fault service time

The evaluation used applications from Rodinia [26], Parboil [27], and Polybench [28] benchmark suites. Table II shows the applications and their access pattern types. The rationale and definition of each type is described in prior work [15]. The memory footprint of these applications vary from 4MB to 130MB with an average of 45MB. The simulation time was too long to evaluate applications with larger footprints. Several applications were not included due to small memory footprint, run-time error, or long simulation time. In some cases, we had to limit the total number of instructions due to excessive simulation time (e.g., *SRD*, *HSD*, etc). We checked that their access patterns were not affected by the limit.

TABLE II: Workload Characteristics

Workload	Abbr.	Footprint	Suite	Access pattern type [15]
hotspot	HOT	12MB	Rodinia	Type I (Streaming Pattern)
leukocyte	LEU	5.6MB		
2DCONV	2DC	128MB	Polybench	
3DCONV	3DC	127.5MB		
backprop	BKP	9MB	Rodinia	Type II (Partly Repetitive Pattern)
pathfinder	PAT	38.5MB		
dwt2d	DWT	27MB		
kmeans	KMN	130MB		
sad	SAD	8.5MB	Parboil	
nw	NW	32MB	Rodinia	Type III (Mostly Repetitive Pattern)
bfs	BFS	37.2MB		
MVT	MVT	64.1MB	Polybench	
BICG	BIC	64.1MB		
srad_v2	SRD	96MB	Rodinia	Type IV (Thrashing Pattern)
hotspot3D	HSD	24MB		
mri-q	MRQ	5MB	Parboil	
stencil	STN	4MB		
heartwall	HWL	40.7MB	Rodinia	Type V (Repetitive-Thrashing Pattern)
sgemm	SGM	12MB	Parboil	
histo	HIS	13.2MB		
spmv	SPV	27.3		
b+tree	B+T	34.7MB	Rodinia	Type VI (Region Moving Pattern)
hybridsort	HYB	104MB		

VI. Evaluation

We first used an unlimited memory capacity to determine the total memory footprint (high watermark) of each application. Next, we reduced the memory size in the simulator to two oversubscription rates: 75% and 50%, so that 75% and 50% of each application's footprint fits in the GPU memory. This section first describes a sensitivity test to determine the threshold values (T1, T2, and T3) for MHPE. Next, an evaluation of CPPE, including a comparison to prior work [11] [16], is presented. Finally, the overhead of CPPE is described and analyzed.

A. Sensitivity Study

MHPE relies on three thresholds to adjust the eviction strategy and forward distance. The thresholds were determined through a sensitivity study. In each simulation, MHPE starts with MRU and an initial forward distance.

T1 (first threshold to switch eviction strategy). We calculated the total untouch level of each interval and selected the maximum value from the first four intervals. The result is shown in Table III. The left most column represents the oversubscription rate. The applications that are not shown had a maximum untouch level of 0. We make three observations. First, applications had a wide range of untouch levels: the highest level was 60 and the lowest level was 0. Second, some applications (e.g., *DWT*, *HWL*) had much different untouch levels at 75% and 50% oversubscription. Third, Type II, III, V, and VI applications had higher untouch levels than Type I and IV applications.

TABLE III: Maximum untouch level in first four intervals

%	DWT	B+T	HIS	BFS	HYB	MVT	NW	HWL
75	60	58	52	51	39	36	32	31
50	0	59	55	45	32	48	16	4
%	KMN	SAD	PAT	SPV	HSD	LEU	SRD	BIC
75	25	22	22	20	14	12	0	0
50	26	38	19	28	14	11	7	0

In Table III, three applications (*HSD*, *LEU*, and *SRD*) favor the MRU eviction strategy. The maximum untouch level for these applications is 14. However, we set T1 to 32 for two reasons. First, 32 is large enough to consider a fluctuation in the untouch level at runtime. This threshold avoids switching the eviction strategy for some regular applications, which favor MRU, with a relatively higher untouch level (e.g., *HSD*). Second, for applications with a medium untouch level (15 $\sim$ 31), MHPE can check the second threshold to adjust the eviction strategy.

T2 (second threshold to switch eviction strategy). To determine this threshold, we removed the applications with a maximum untouch level larger than 32 (see Table III). The total untouch level was calculated in the first four intervals for the remaining applications. The result is shown in Table IV, which has a similar trend to Table III. To differentiate *HSD* from applications that favor LRU, we set T2 to 40. Using the selected values for T1 and T2, most irregular applications switch their eviction strategy to LRU by the end of the fourth interval.

TABLE IV: Total untouch level in the first four intervals

%	KMN	PAT	HWL	SAD	SPV	NW	HSD	LEU	SRD	DWT
75	70	67	59	46	45	/	37	12	0	/
50	58	60	4	/	68	38	30	21	7	0

T3 (forward distance limit). To determine this threshold, we considered *SRD*, *HSD*, and *MRQ* because they undergo continuous adjustment at runtime. We tested candidate values from 16 to 40 with a stride of 4. A limit value of 32 had the best average performance, and thus, we set T3 to 32.

MHPE divides the untouch level (0 $\sim$ 31 when T1 is 32) into five ranges to determine forward distance: [0 $\sim$ 3], [4 $\sim$ 10], [11 $\sim$ 17], [18 $\sim$ 24], and [25 $\sim$ 31]. When an interval is over, MHPE calculates the untouch level, compares it to the number of wrong evictions, and selects the larger value, and adds the value to the current forward distance. This mechanism ensures the forward distance is adjusted gradually.

B. Performance Evaluation

To evaluate the performance of CPPE, we did several experiments. First, we evaluated the two deletion schemes introduced in Section IV-C. Second, we compared CPPE to a state-of-the-art baseline, which combines LRU and a locality prefetcher [9] [11]. When GPU memory fills to capacity, the prefetcher continues to prefetch a whole chunk of pages. This baseline is similar to prior work [16]. Third, we explored whether the baseline's inefficiency can be addressed by changing the eviction policy. In the experiments, we evaluated Random and reserved LRU [16] with the same prefetch strategy as the baseline. Finally, we evaluated how disabling prefetch under memory oversubscription affected performance.

Pattern Deletion Schemes As described earlier, we propose two schemes to delete touch patterns from the pattern buffer (the pattern is a bit vector that records the touched pages in a chunk). The first scheme (Scheme-1) deletes a touch pattern whenever a faulted page does not match the pattern, and the second scheme (Scheme-2) deletes a pattern when a faulted page does not match the pattern during the first search.

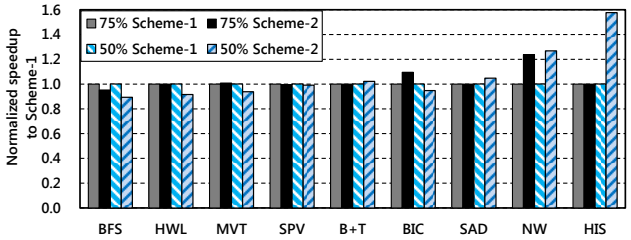

Fig. 7: Comparison of pattern deletion scheme.

Fig. 7 compares the performance of the two schemes, which perform similarly for *MVT*, *SPV*, *B+T*, *BIC*, and *SAD*. The outliers are *BFS*, *NW*, *HWL*, and *HIS* (50% oversubscription). Through further analysis, we observed that the recorded chunks of *NW* and *HIS* had specific patterns (e.g., touched pages have a fixed stride), and the chunks of *BFS* and *HWL* usually needed a long time to be fully populated.

Scheme-2 performed better for applications that had specific patterns within a chunk because these chunks were kept in the buffer once the pattern is detected. In contrast, chunks that needed a long time to be fully populated favored Scheme-1, because they were removed from the buffer once a mismatch was detected. After that point, the whole chunk was migrated to the GPU memory with only one prefetch. However, Scheme-2 usually required two prefetches for the chunk to be fully migrated. On average, Scheme-2 achieved 3% and 7% performance improvement for 75% and 50% oversubscription, respectively. Due to its performance advantage, we adopt Scheme-2 in the following evaluation.

Comparison to baseline. Fig. 8 shows a comparison of CPPE to the baseline. *MVT* and *BIC* are omitted because they crashed in the baseline. We make three observations. First, CPPE performed similarly to the baseline for Type I and VI applications, which favor LRU. Second, CPPE outperformed the baseline significantly for Type IV applications, which suffered with LRU. Third, CPPE's access pattern-aware prefetcher had a performance advantage over the naïve prefetcher, especially for severely thrashing applications, such as *SAD*, *HIS*, and *NW*.

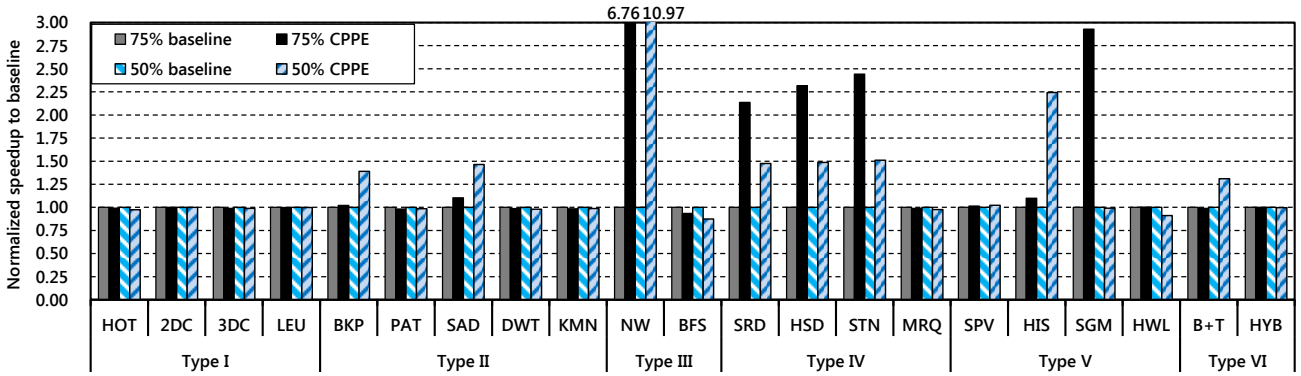

Fig. 8: Performance of CPPE normalized to baseline.

With CPPE, *MVT* and *BIC* run to completion and experience less thrashing. On average, CPPE achieved 1.56x and 1.64x speedup over the baseline at 75% and 50% oversubscription.

CPPE performed slightly worse than the baseline for *BFS* and *HWL* (at 50% oversubscription) with Scheme-2. Using Scheme-1, CPPE performed similarly to the baseline for these applications. In addition, CPPE had no performance benefit for *MRQ* because the forward distance was continuously adjusted due to wrong evictions, which reduced MRU's effectiveness.

Comparison to other mechanisms. We also evaluated other eviction policies, including Random and reserved LRU. For comparison, these policies were combined with the naïve prefetcher. The result is shown in Fig. 9. In this figure, LRU-10%/LRU-20% means that the top 10%/20% chunks in the LRU chain are reserved. We make three observations. First, reserving the top percentage of chunks in the LRU chain improved LRU's performance for applications with thrashing access patterns (e.g., Type IV and V). However, the performance improvement was related to the reservation percentage. This value is difficult to determine in advance. Even with a high reservation percentage (e.g., 20%), reserved LRU did worse than CPPE. It was also worse than Random. Second, reserved LRU hurt performance for some LRU-friendly applications when they were sensitive to memory capacity. For example, on average, LRU-10% suffered from 27% performance loss for Type VI applications under 50% oversubscription. Third, simply changing eviction policy may not address the inefficiency of the baseline. In contrast, CPPE performed better than or similarly to these policies for all access pattern types, which demonstrates CPPE's effectiveness.

Comparison to disabling prefetching under oversubscription. We observed that disabling prefetches when the GPU memory fills to capacity causes severe (up to 87%) performance slowdown for regular applications. To determine whether this method works for applications that thrash in the baseline (see Section III), we compared disabling prefetching against the baseline and CPPE. The result is shown in Fig. 10.

As *MVT* and *BIC* crashed in the baseline (denoted by 'X' in the figure), we normalized CPPE's performance to this

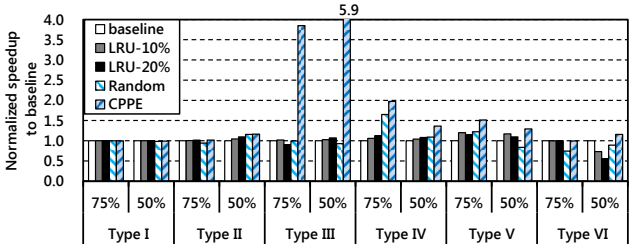

Fig. 9: Comparison of other prior work to CPPE.

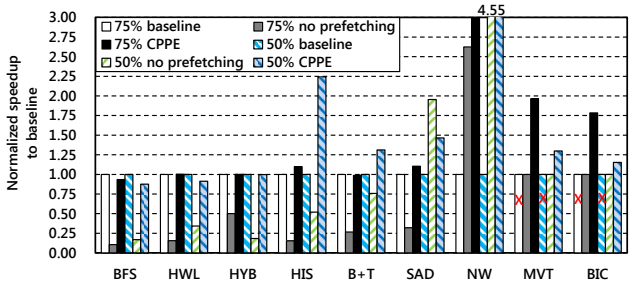

Fig. 10: Performance of disabling prefetch when memory full.

method. We make three observations. First, for applications that had less thrashing, disabling prefetching incurred significant (up to 85%) performance slowdown. Second, disabling prefetching had a performance improvement for *SAD* (at 50% oversubscription), *NW*, *MVT*, and *BIC*. These applications severely thrash in the baseline. Third, CPPE substantially outperformed turning off the prefetcher under memory oversubscription, even for *MVT* and *BIC*, which crashed in the baseline. CPPE was worse than disabling prefetching for only *SAD*. A specific touch pattern did not manifest in evicted chunks for this application. *SAD* is capacity sensitive with high oversubscription (e.g., 50%). These characteristics make disabling the prefetcher a better choice in this case.

C. Overhead Analysis

CPPE uses three structures for page prefetch and eviction: a chunk chain, a buffer to record touch patterns (used by the

prefetcher), and a buffer to store evicted chunks in the previous few intervals (used to adjust forward distance). Each chunk has an entry in the chunk chain and the entry has two fields: tag and data. The tag is the chunk address and the data is a bit set that records whether a page in the chunk has been touched. Assume the system is 64-bit wide and the page size is 4KB, and thus, the tag is 48 bits. In fact, a data type that is 8B (64 bits) can be used to record chunk address. As chunk size is 16, a bit set that has 16 bits is enough, which is 4B (32 bits) in C++. Therefore, each entry needs 12B. For simplicity, we assume all three structures use the same entry. We first calculate the total number of entries needed for these three structures, and then calculate the average of number of entries for our benchmarks. Results show that, on average, 731 entries and 559 entries are needed under oversubscription rates of 75% and 50%. This translates to 8.6KB and 6.6KB storage cost in the GPU driver. As the structures are stored in the CPU memory, this overhead is negligible.

Compared to HPE, MHPE has less overhead. First, prefetching pages reduces the frequency of updates to the chunk chain. For example, a chunk needs 16 updates with HPE, and only one update with MHPE. Second, the counter field is removed from the chunk chain entry in CPPE. This simplifies MRU-C (HPE) to MRU (MPHE), which eliminates search overhead to select an eviction candidate from the MRU position. Although MHPE calculates the untouch level to adjust the eviction strategy and forward distance, this operation involves simple calculation and comparison, introducing minimal overhead.

Compared to the baseline, CPPE introduces two buffers to record touch pattern and evicted chunks. However, these buffers incur acceptable overhead. The average buffer (used to record evicted chunks) length for MHPE is 73 and 51 at 75% and 50% oversubscription, which is smaller than the chunk chain length. In addition, searching the buffer is not on the critical path of handling a page fault. To estimate the overhead related to the touch pattern buffer, we calculated the average buffer length of applications that used the buffer at runtime. We divide the buffer length by chunk chain length and use the normalized percentage as the metric. Results show that the average buffer length was 37.2% and 88.7% of the chunk chain length for oversubscription of 75% and 50%, respectively. This overhead is acceptable for a few reasons. First, the buffer is used in limited cases; that is, when the eviction strategy is switched to LRU and evicted chunks have an untouch level larger than 8. Second, a search of the buffer is not on the critical path of page fault handling, and the search can be done in parallel with the search of the chunk chain. Third, prefetching pages significantly reduces the search frequency, and hence, incurs less overhead.

VII. Related Work

To our knowledge, this paper is the first to propose a fine-grained mechanism to coordinate page prefetch and eviction to manage memory oversubscription for GPUs with unified memory. Previous research aims to 1) provide efficient address translation on GPUs, 2) address the inefficiency of unified memory, and 3) manage memory oversubscription in GPUs with unified memory.

GPU Address Translation. Unified memory requires support for address translation. J. Power *et al.* [18] and B. Pichai *et al.* [19] were among the first to explore GPU address translation. J. Power *et al.* showed that per-SM post-coalescing TLBs, a highly-threaded page table walker, and a shared page walk cache are essential for efficient address translation. B. Pichai *et al.* showed the importance of making the warp scheduler to be aware of the TLB. R. Ausavarungnirun *et al.* [17] demonstrated the performance advantage of a shared L2 TLB over a shared page walk cache. J. Vesely *et al.* [25] analyzed shared virtual memory for on-die GPUs using real system measurements. They found that serving TLB misses and page faults from the GPU is much slower than serving them from the CPU. Our work adopts designs from [17] and [18] for unified memory infrastructure, including per-SM private L1 TLBs, a shared L2 TLB, a highly-threaded page table walker, and a shared page walk cache.

Addressing Inefficiency of Unified Memory. To reduce stalls due to page faults, T. Zheng *et al.* [9] proposed *replayable far faults* to enable SMs to continue execution in the presence of page faults. To support efficient address translation for irregular applications, S. Shin *et al.* [29] optimized the order of page walks by prioritizing translation requests from instructions that require less work to service their address translation needs. They also proposed mechanisms to coalesce address translation of all pending page table walkers in the same neighborhood that happen to have their address mappings fall on the same cache line [24]. To support multiple page sizes in GPUs, R. Ausavarungnirun *et al.* [17] proposed *Mosaic*, a GPU memory manager that uses base pages to transfer data over the system I/O bus, and allocates physical memory in a way that preserves base page contiguity and ensures that a large page frame contains pages from only a single memory protection domain. To support multi-application concurrency in GPUs, they also proposed *MASK* [20], a new GPU framework that provides low-overhead virtual memory support for the concurrent execution of multiple applications.

GPU Memory Oversubscription Management. T. Zheng *et al.* [9] proposed a locality prefetcher to mitigate the performance loss caused by expensive page faults. They also evaluated the performance of LRU and Random policy, and studied performance sensitivity to oversubscription. Q. Yu *et al.* proposed hierarchical page eviction [15]. HPE dynamically maintains a chunk chain. It uses statistics to classify applications into three categories and selects an appropriate eviction strategy for each category. HPE addresses LRU's inability to handle thrashing access patterns while retaining LRU's advantages for LRU-friendly patterns. However, HPE does not support prefetching. D. Ganguly *et al.* showed that making the page eviction policy to be aware of prefetching semantics is effective [16]. They also proposed reserved

LRU to handle thrashing access patterns. However, we found that incorporating prefetcher and page eviction in a coarse-grained manner works poorly for some irregular applications. In addition, reserved LRU hurts performance of some irregular applications. CPPE incorporates a page prefetcher and eviction policy in a fine-grained manner. This is different from prior work, which either focuses on page prefetching or eviction. Compared to HPE, CPPE uses a modified eviction policy to support prefetching. Compared to D. Ganguly *et al.*'s work [16], CPPE coordinates a more efficient eviction policy (MHPE rather than reserved LRU), and a smarter prefetcher that prefetches pages according to detected touch pattern. C. Li *et al.* [11] proposed a hardware/software cooperative solution for memory oversubscription. The main mechanisms (memory-aware throttling and capacity compression) are hardware schemes that require modifications to the GPU hardware. These hardware mechanisms are orthogonal to our software-based solution.

VIII. Conclusion

In this paper, we propose CPPE to manage memory oversubscription for GPUs with unified memory. CPPE incorporates a modified HPE policy (MHPE) and an access pattern-aware prefetcher. MHPE is aware of the prefetcher's semantics: it selects a chunk to evict that was prefetched. The prefetcher is also aware of MHPE: it prefetches pages according to the touch pattern in evicted chunks that were selected by MHPE. Simulation results are promising: CPPE has a 1.56x and 1.64x (up to 10.97x) speedup over a state-of-the-art software baseline, which combines a naïve prefetcher and LRU policy. CPPE outperforms other mechanisms, including Random and reserved LRU policies coupled with the naïve prefetcher. It also performs better than simply disabling prefetching under memory oversubscription.

Acknowledgment

We thank the anonymous reviewers for their feedback. This work was supported in part by National Natural Science Foundation of China (Grants 61872374, 61433019, and 61572508).

References

[1] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *ISCA*, 2017, pp. 320–332.

[2] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *MICRO*, 2018, pp. 339–351.

[3] NVIDIA, *GPU-Accelerated Applications*, 2016, http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[4] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence," in *HPCA*, 2016, pp. 494–506.

[5] M. Harris, "Unified Memory in CUDA 6," 2013, https://devblogs.nvidia.com/unified-memory-in-cuda-6/.

[6] N. Sakharnykh, "Beyond GPU Memory Limits with Unified Memory on Pascal," 2016, https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.

[7] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *HPCA*, 2015, pp. 354–365.

[8] NVIDIA, "Compute Unified Device Architecture," 2014, https://developer.nvidia.com/cuda-zone.

[9] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards High Performance Paged Memory for GPUs," in *HPCA*, 2016, pp. 345–357.

[10] N. Sakharnykh, "Unified Memory on Pascal and Volta," 2017, http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf.

[11] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A Framework for Memory Oversubscription Management in Graphics Processing Units," in *ASPLOS*, 2019, pp. 49–63.

[12] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007, pp. 381–391.

[13] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *ISCA*, 2010, pp. 60–71.

[14] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "Hierarchical Page Eviction Policy for Unified Memory in GPUs," in *ISPASS*, 2019, pp. 149–150.

[15] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, doi: 10.1109/TCAD.2019.2944790.

[16] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory," in *ISCA*, 2019, pp. 224–235.

[17] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes," in *MICRO*, 2017, pp. 136–150.

[18] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA*, 2014, pp. 568–578.

[19] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014, pp. 743–758.

[20] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency," in *ASPLOS*, 2018, pp. 503–518.

[21] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping," in *VEE*, 2015, pp. 65–77.

[22] NVIDIA, "NVIDIA Tesla P100," 2016, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[23] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009, pp. 163–174.

[24] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-Aware Address Translation for Irregular GPU Applications," in *MICRO*, 2018, pp. 352–363.

[25] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems," in *ISPASS*, 2016, pp. 161–171.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009, pp. 44–54.

[27] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[28] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-Tuning a High-Level Language Targeted to GPU Codes," in *IPC*, 2012, pp. 1–10.

[29] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling Page Table Walks for Irregular GPU Applications," in *ISCA*, 2018, pp. 180–192.