The Journal of Supercomputing (2020) 76:2958–2985
https://doi.org/10.1007/s11227-019-03079-y

A quantitative evaluation of unified memory in GPUs

Qi Yu[1] · Bruce Childers[2] · Libo Huang[1] · Cheng Qian[1] · Zhiying Wang[1]

Published online: 16 November 2019

Abstract

The introduction of unified memory and demand paging has simplified programming of graphics processing units (GPUs). It has also enabled oversubscribing the memory for a GPU. However, the overhead of page management makes page faults a performance bottleneck. Sometimes the page eviction policy is unable to mitigate performance slowdown caused by page faults and memory oversubscription. On average, eviction policies such as Random and CAR are not competitive with a traditional least recently used (LRU) policy. Other policies, such as CLOCK-Pro, are designed to overcome challenges with LRU, but they only achieve limited speedup. Even enhancing LRU with page walk hit information does not lead to notable performance improvement. Based on these observations, we propose optimization opportunities to mitigate performance degradation caused by page faults and memory oversubscription. These optimization opportunities include an effective page eviction policy that retains LRU's advantages while addressing LRU's inability to deal with thrashing access patterns, page prefetch and pre-eviction, memory-aware throttling, and capacity compression.

Keywords GPUs · Unified memory · Performance evaluation · Memory oversubscription · Eviction policy

✉ Libo Huang
libohuang@nudt.edu.cn

Qi Yu
yuqi13@nudt.edu.cn

Bruce Childers
childers@pitt.edu

Cheng Qian
qiancheng@nudt.edu.cn

Zhiying Wang
zywang@nudt.edu.cn

[1] School of Computer, National University of Defense Technology, Changsha, Hunan, China

[2] University of Pittsburgh, Pittsburgh, PA, USA

1 Introduction

Due to massive thread-level parallelism, graphics processing units (GPUs) have evolved from dedicated accelerators to be mainstream commodity computing devices. Recent support for *unified memory* [14] and *demand paging* [35] further boosts the adoption of GPUs in many application domains. This innovation simplifies programming and eases the porting of applications to GPUs by providing a single, unified virtual address space to access all CPU and GPU memory in the system [29].

In early versions of GPU programming models, such as CUDA [28] and OpenCL [22], programmers had to manage memory allocation and CPU–GPU data transfers using an application programming interface (API). This was challenging because the data shared between the CPU and GPU required two allocations, and the data had to be transferred to the right place at the right time under control of the programmer. This type of "explicit memory management" by the programmer is, especially, challenging for *memory oversubscription*, where dataset size exceeds the GPU memory capacity. In this case, the programmer has to divide the dataset into pieces (e.g., overlays) and move them under program control, which is error-prone and time-consuming. Recent support for *memory virtualization* in GPUs, including unified memory and demand paging, has simplified the programming model with "implicit memory management." A software-managed runtime (the CUDA/OpenCL runtime) allows faulted pages to be automatically migrated to the GPU memory on-demand. This feature provides several benefits. First, it eliminates data copying between the CPU and GPU, which makes it easier to write GPU programs and simplifies porting existing applications to the GPU. Second, implicit memory management enables overlapping data transfer and GPU execution in a transparent way by allowing kernel launch before any data transfer has occurred. Third, it enables memory oversubscription, that is, the GPU can compute across datasets that exceed memory capacity without explicit (manual) management. Unfortunately, the benefits of unified memory come at a cost. As current GPUs do not support precise exceptions [21], the manipulation of the GPU page table and data transfers is offloaded to a software runtime system (the GPU driver) on the host CPU. In this model, a page fault leads to several round trips over PCIe and interaction with the host CPU, which has high overhead (e.g., 20 or more microseconds [36, 44]).

To better understand how performance is influenced by unified memory and memory oversubscription, this paper describes a quantitative evaluation and analysis of unified memory utilizing a popular GPU simulator–GPGPU-Sim [3]. We first explore typical access patterns in GPU applications from three benchmarks: Rodinia [5], Parboil [40], and Polybench [12]. Then we test the sensitivity of applications to page fault latency. Next, we analyze how oversubscription impacts performance. Finally, we evaluate four eviction policies: LRU, Random, CAR [4], and CLOCK-Pro [19]. All four policies, except Random, track all page walk information (i.e., "perfect information"). In addition, we compare the performance of perfect LRU and real LRU, which only tracks page fault information.

We make five observations:

- GPU applications have various access patterns. We find there are six representative access patterns, including regular ones and irregular ones.
- Page faults dominate the time consumed in the memory hierarchy, and hence, this is the performance bottleneck.
- Oversubscription has a significant influence on performance. Most applications experience severe performance degradation when the GPU memory is 25% oversubscribed. Sometimes, it is beyond the capability of an eviction policy to mitigate performance slowdown.
- On average, both Random and CAR are not competitive with LRU, and CLOCK-Pro only achieves limited speedup over LRU, despite outperforming LRU in some cases.
- Real LRU is competitive with perfect LRU on average, and hence, suppressing the need to enhance LRU with page walk hit information.

Based on these observations, we identify optimization opportunities to mitigate performance degradation caused by long-latency page faults and memory oversubscription. For applications with regular access patterns, adaptive prefetching and pre-eviction work well; for applications with irregular access patterns, memory-aware throttling and capacity compression are effective to alleviate thrashing due to limited memory capacity. Finally, an effective eviction policy that retains LRU's advantages while addressing LRU's inability to handle thrashing access patterns is necessary for GPUs with unified memory.

The rest of this paper is organized as follows. Section 2 introduces background on GPU architecture and unified memory. Section 3 motivates the need for a quantitative evaluation of unified memory. Section 4 describes the methodology, including experimental setup and benchmarks used for evaluation. Section 5 summarizes typical access patterns in GPU applications. Section 6 analyzes results of application sensitivity to page fault latency. Section 7 analyzes results of memory oversubscription and eviction policies. Section 8 describes the optimization opportunities. Section 9 discusses related work, and Sect. 10 concludes this paper.

2 Background

Several details of GPU architecture are important to analyze performance, including computing units, the memory hierarchy, and the execution model. The way address translation works is also important and a potential source of performance overhead.

2.1 GPU architecture

A GPU has multiple *streaming multiprocessors* (SMs), also known as *shader cores*. Each SM has a set of *shader processors* (SPs, by NVIDIA) or *stream processors* (by AMD) that execute integer and float-pointing instructions, special function units

(SFU) for complex arithmetic instructions, and load-store units (LDST). A group of 32 (64) threads, called a *warp* (*wavefront*), share the same program counter (PC) and execute the same instruction at the same time, which is known as *single instruction multiple threads* (SIMT) execution model [10, 26]. Under SIMT, all threads in a warp execute in lockstep. Long-latency memory accesses are tolerated via *fine-grained multithreading*: the GPU swaps out warps that stall on memory for other ready warps, which can then execute.

Modern GPUs typically have multiple levels of memory. Each SM has its own private L1 data cache, read-only texture cache, constant cache, and software-managed scratchpad memory, named *shared memory* by NVIDIA or *local memory* by AMD. Each memory partition has a slice of L2 cache and a memory controller that is shared by SMs. The memory partitions and SMs are connected via on-chip network. More details about GPU memory hierarchy can be found in [15, 20, 27].

2.2 Unified memory

Memory virtualization in GPUs requires address translation, such as CPU address virtualization. Although the details of the memory hierarchy of common GPU architectures from NVIDIA, AMD, and Intel have not been published, it is accepted that they support translation-lookaside buffers (TLBs) for fast address translation [1]. Previous work [32] showed that per-SM post-coalescing L1 TLBs, highly threaded page table walker, and a page walk cache are essential components for efficient address translation. Figure 1 depicts a commonly used GPU architecture with support for address translation in recent research [1]. This architecture replaces the shared page walk cache with a shared L2 TLB because the latter has better performance [1]. In this figure, each SM has a private L1 TLB, which is shared across all SPs. These per-SM L1 TLBs are backed by an L2 TLB, which is shared by all SMs. The highly threaded page table walker supports multiple concurrent walks.

Before accessing L1 cache, an attempt is made to first translate a memory address by checking the L1 TLB for a virtual-to-physical address translation (see ❶ Fig. 1). On a miss, the shared L2 TLB is accessed (❷). If the request also misses in the shared L2 TLB, the page table walker is invoked to begin a walk (❸). The walker accesses each level of the page table, retrieving the desired data either from the shared L2 cache or from GPU main memory (❹). If the walker fails to find the address translation in the page table, a *page fault* is incurred. In this case, the GPU stops address translation in the faulted SM, and notifies the CPU that there is a page fault. An interrupt is raised in the CPU, and the operating system performs a page table walk. The CPU unmaps the page from its page table and migrates the page to the GPU. Once the page fault is serviced, the GPU retries the address translation for the faulting address. If the GPU memory fills to capacity, before serving the page fault, the GPU driver selects an eviction candidate and notifies the GPU to migrate the evicted page to the CPU. Once all of the in-flight requests that could have been accessing this evicted page are resolved, the page is migrated to the CPU and unmapped from the GPU's page table [6]. In addition, all TLB and cache entries related to the evicted page are invalidated or flushed.

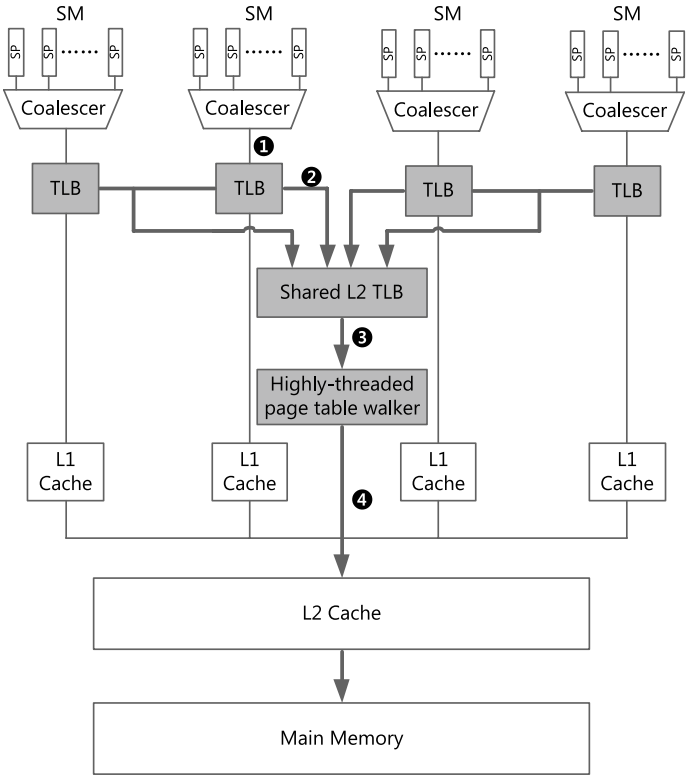


Fig. 1 Baseline GPU architecture with support for address translation

3 Motivation

Page faults and memory oversubscription are two main sources of long-latency memory operations in unified memory. Understanding how they influence performance sheds insight on how best to optimize the GPU design for memory virtualization. Although there has been some research studying unified memory, the prior work mainly focused on proposing optimization techniques to address the inefficiency of unified memory rather than a quantitative evaluation, such as [24, 44]. In this paper, we present a quantitative evaluation and comprehensive analysis of unified memory by answering the following fundamental questions about page faults and memory oversubscription:

1. *What are the typical access patterns in GPU applications?* The *access pattern* is the manifestation of the access behavior of an application. It may influence performance because different patterns can react differently to memory oversubscription. In this paper, we focus on the access pattern of page walks.
2. *How does page fault latency impact performance?* The amount of time it takes to migrate a missing page to the GPU memory is the *page fault latency*. Studying

its influence is important because (1) we need to determine whether reducing this latency is necessary, and (2) what aspects should optimization techniques target. Although prior work [24, 44] has done a sensitivity test of page fault latency, these studies did not examine the difference (in terms of sensitivity) among GPU applications and what factors contribute to performance difference.

3. *How does memory oversubscription influence performance and how does performance change under different amounts of oversubscription?* Memory oversubscription requires page management, i.e., page eviction. This is another long-latency operation. In the era of big data, GPUs tend to process datasets that exceed their memory capacity. As a result, studying the influence of memory oversubscription is meaningful to decide how to optimize page management.
4. *How do common eviction policies perform, and what are the pros and cons of these policies for GPU applications?* Eviction policies have a direct impact on performance because they determine the pages to be evicted. This necessitates a quantitative evaluation of commonly used eviction policies in GPUs with unified memory. Understanding their advantages and disadvantages is useful in designing more effective eviction policies with high performance and low overhead.

4 Methodology

To examine the above questions, we use GPGPU-Sim as the experimental tool because it permits varying key parameters, such as page fault latency. This section describes our experimental methodology, including the baseline architecture and configurations, as well as our benchmarks.

4.1 Experimental setup

We modified GPGPU-Sim 3.2.2 with TLBs and a GPU MMU (Memory Management Unit) [31, 32] to simulate unified memory. We adopted a two-level TLB design, which includes a private per-SM L1 TLB and an L2 TLB shared across all SMs (similar to [1, 2, 24, 38, 39]). The page table walker allows up to 64 concurrent walks, which is similar to [1, 24]. A single-level page table and a fixed page walk latency (20 cycles) were used to simplify simulation. We also used a fixed latency (20 µs) to model page fault handling and eviction (same as prior work [44]). We implemented replayable far-faults [44] to enable an SM to continue execution in the presence of page faults. We assumed the GPU uses 4KB OS pages. The default system configurations are shown in Table 1.

4.2 Benchmarks

We examined access patterns and did a quantitative evaluation of unified memory using applications from Rodinia [5], Parboil [40], and Polybench [12]. We tested all applications from these benchmark suites. However, some applications are not included in the results due to small memory footprint, long simulation time, runtime

Table 1 Configurations of the simulated system

GPU cores	15 Cores, 1.4 GHz, GTO scheduler [34]
Private L1 cache	16 KB, 4-way associative, LRU
Private L1 TLB	128-Entry per SM, single port, 1-cycle latency, LRU, support hit under miss
Shared L2 cache	1.5 MB total, 128 KB/DRAM channel, 8-way associative, LRU
Shared L2 TLB	512-Entry, 16-associative, LRU, 10-cycle latency, 2 ports
Page table walker	Shared page table walker, 64 concurrent walks
DRAM	GDDR5, 12-Channel, FR-FCFS scheduler, 177.4 GB/s aggregate
CPU–GPU interconnect	16 GB/s, 20 µs page fault service time

error in the simulator, or similar access pattern as other applications. The details of the selected applications are shown in Table 2. The memory footprints of these applications vary from 3 to 130 MB, with an average of 35.7 MB. For some benchmarks, we had to limit the number of instructions (*SRD*, *HSD*, etc.) to avoid excessively long simulation time. The number of instructions was limited without affecting access pattern. A few benchmarks had very large memory footprints, which we could not simulate.

5 Exploring access patterns

We used default configurations with unconstrained memory capacity to record all page walk traces. Then we analyzed the access pattern of each application using its trace. We ran simulations multiple times and found that the application's access pattern is the same in each simulation. We also recorded the number of page faults and page walks (including page walk hits and page faults) for each application. To better understand the relation between the number of page walks and the number of page faults, we define a metric called *page walk-fault ratio (PWFR)*, which is calculated as the number of page faults divided by the number of page walks. We identified six representative access patterns in the benchmarks. For each access pattern, we selected a representative application and plotted the pattern.

Streaming Access Pattern [17] *(Type I)*. This is a simple and regular access pattern, where all pages are touched only once. As a result, the pattern has no hits under any eviction policy. *HOT*, *LEU*, *LMD*, *CUT*, *GEM*, and *2DC* have this pattern.

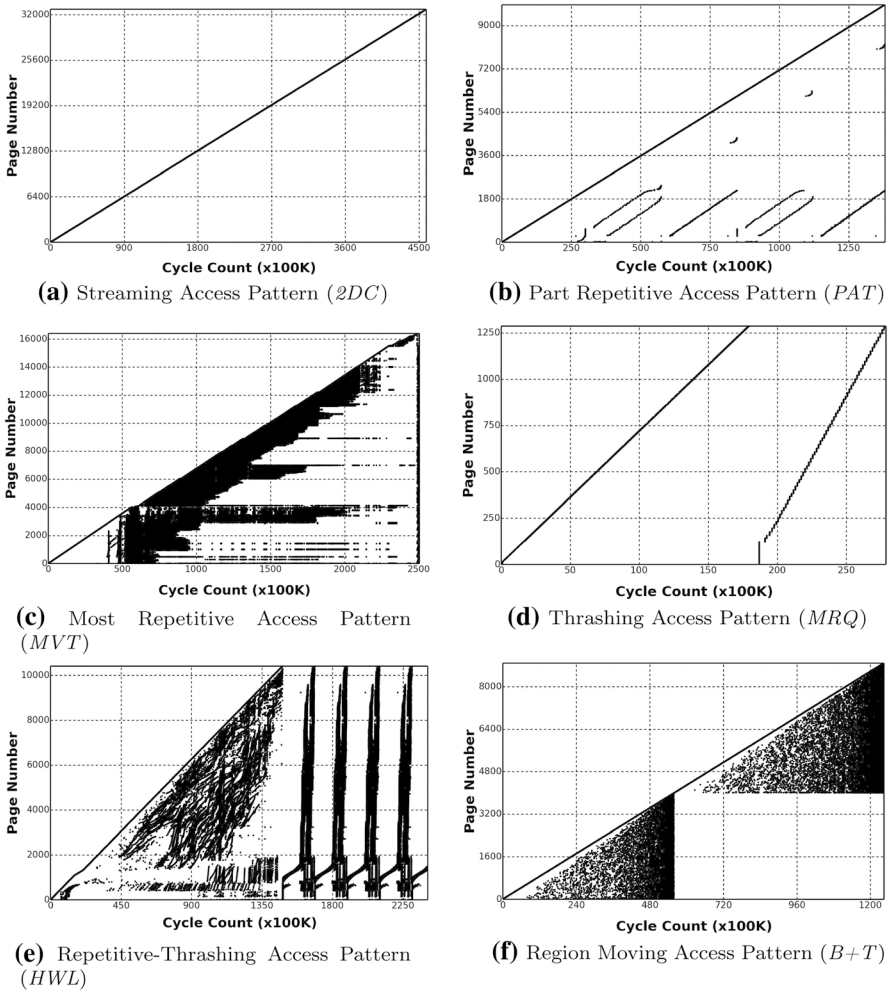
Table 2 Workload characteristics

Benchmark suite	Application and abbreviation
Rodinia	hotspot (HOT), backgrop (BKP), pathfinder (PAT), dwt2d (DWT), nw (NW), k means (KMN), b+tree (B+T), hybridsort (HYB), bfs (BFS), hotspot3D (HSD), srad_v2 (SRD), heartwall (HWL), leukocyte (LEU), lavaMD (LMD)
Parboil	cutcp (CUT), sad (SAD), mri-q (MRQ), stencil (STN), sgemm (SGM), histo (HIS), spmv (SPV)
Polybench	2DCONV (2DC), GEMM (GEM), 3MM (3MM), MVT (MVT), BICG (BIC)

Figure 2a depicts the access pattern of *2DC*. As all pages are touched once, the number of page walks is equal to that of page faults. Therefore, the PWFR of these applications is 1, which can be seen in Table 3.

Part Repetitive Access Pattern (Type II). Some pages have re-reference with some probability after the initial page fault. The reference frequency is not uniformly distributed, with some pages receiving frequent touches and other pages receiving infrequent touches. *PAT*, *KMN*, *BKP*, *DWT*, *SAD*, and *3MM* have this pattern. Figure 2b shows the access pattern of *PAT*. Although these applications have a difference in their specific patterns, the PWFR is typically smaller than 2 (cf. Table 3).

Most Repetitive Access Pattern (Type III). These applications differ from Type II—most pages are touched multiple times following the initial fault. Some pages



(a) Streaming Access Pattern (*2DC*)

(b) Part Repetitive Access Pattern (*PAT*)

(c) Most Repetitive Access Pattern (*MVT*)

(d) Thrashing Access Pattern (*MRQ*)

(e) Repetitive-Thrashing Access Pattern (*HWL*)

(f) Region Moving Access Pattern (*B+T*)

Fig. 2 Six representative access patterns

Table 3 Access patterns and corresponding applications

Access pattern	Benchmark	Application	PWFR
Streaming access pattern (Type I)	Rodinia	HOT	1.00
		LEU	1.00
		LMD	1.00
	Parboil	CUT	1.00
	Polybench	GEM	1.00
		2DC	1.00
Part repetitive access pattern (Type II)	Rodinia	PAT	1.09
		KMN	1.15
		BKP	1.50
		DWT	1.67
	Parboil	SAD	1.75
	Polybench	3MM	1.29
Most repetitive access pattern (Type III)	Rodinia	NW	67.51
		BFS	251.64
	Polybench	MVT	787.71
		BIC	835.94
Thrashing access pattern (Type IV)	Rodinia	SRD	3.00
		HSD	9.18
	Parboil	MRQ	2.00
		STN	41.85
Repetitive-thrashing access pattern (Type V)	Rodinia	HWL	12.81
	Parboil	SGM	36.40
		HIS	18.78
		SPV	9.36
Region moving access pattern (Type VI)	Rodinia	B+T	4.15
		HYB	349.73

are even touched tens of times. *NW*, *BFS*, *MVT*, and *BIC* have this pattern. Figure 2c depicts the access pattern of *MVT*. Although these applications have different patterns, they typically have very large PWFR (cf. Table 3). *NW* has the smallest PWFR value; *MVT* and *BIC* have a very large PWFR (over 780). This demonstrates that the majority of pages are referenced very frequently. Therefore, these applications are usually sensitive to oversubscription, because some pages may incur significant thrashing if memory footprint does not fit in the GPU memory.

Thrashing Access Pattern [17] *(Type IV)*. This is a regular access pattern because all pages are referenced *N* times. *SRD*, *HSD*, *MRQ*, and *STN* have this pattern. Figure 2d shows the access pattern of *MRQ*. For these applications, PWFR is related to the amount of repeat (*N*), with more repeat leading to larger PWFR. We can see from Table 3 that *HSD* and *STN*'s PWFR is not an integer. The number of instructions was limited in these applications by our simulation methodology, which leads to pages being referenced not the same times. The difference (in terms of repeat)

among these applications is significant with *MRQ* repeating only twice and *STN* repeating more than 41 times.

Repetitive-Thrashing Access Pattern (Type V). This pattern is a combination of Type III and Type IV: the sequence repeats N times, and in each iteration, most pages are referenced multiple times. The PWFR relies on two factors: the repeat time N and the fraction of pages that are touched multiple times in one iteration. *HWL*, *SGM*, *HIS*, and *SPV* have this pattern. Figure 2e depicts the access pattern of *HWL*. From Table 3, these applications have moderate PWFR. Compared to applications of Type III, pages with this pattern are usually referenced less frequently in one iteration.

Region Moving Access Pattern (Type VI). Pages are usually split into several address regions, and in each region, pages are referenced multiple times for a certain period of time. The application continues to access pages in the next region, after which time, pages in the previous regions will not be touched again. The PWFR of these applications relies on two factors: the number of address regions and the reference density in each region. We find that only two applications have this pattern: *B+T* and *HYB*. Figure 2f shows the access pattern of *B+T*. From Table 3, the PWFR of these applications varies significantly. *B+T* has a PWFR smaller than 5, while *HYB* has a PWFR value larger than 300.

Summary. We find six representative access patterns in our set of 26 applications. Types I, II, and IV may be regarded as regular patterns because they are easily predicted, especially for Types I and IV. Types III, V, and VI are irregular access patterns because pages are usually referenced different times and the references are not easily predicted. These access patterns differ significantly in their specific patterns and PWFR. Even within the same access pattern, applications may have different PWFR, e.g., applications of Type III and Type VI. In addition, these access patterns react differently to oversubscription and the page eviction policy. More details can be found in Sects. 6 and 7.

6 Sensitivity to page fault latency

We evaluated application sensitivity to page fault latency, which is the time to service a page fault. To simplify the design, we used a fixed page fault latency in the evaluation. Each time, we used default configurations with unconstrained memory capacity. We changed page fault latency and left the other parameters unchanged.

According to prior work [44], the page fault penalty ranges from 20 to 50 µs. For a simple scheme where the faulting process merely uses DMA to copy data to the GPU, the overhead may be on the low end of this range (20 µs). In a complex situation, where the software fault handler performs significant CPU and GPU page table manipulations on the host, before invalidating and updating the GPU resident page table, the overhead is more expensive (50 µs). Thus, we tested four candidate values for page fault latency: 20–50 µs with a step of 10 µs. Besides, we also tested an unrealistically low latency of 10 µs to explore the upper bound of performance. We used instructions per cycle (IPC) as the evaluation metric. Performance is normalized to the case with page fault latency equal to 10 µs. If page faults are the bottleneck, the performance degradation should be

roughly proportional to the increase in page fault latency. More specifically, the normalized IPC should be around 0.5, 0.33, 0.25, and 0.2 when page fault latency is 20 µs, 30 µs, 40 µs, and 50 µs, respectively.

Figure 3 shows application sensitivity to page fault latency. Most applications follow the expected trend exactly or roughly. However, there are notable outliers: *CUT*, *LEU*, *LMD*, *MRQ*, and *HWL*. *LEU* and *LMD* are insensitive to page fault latency with performance degradation less than 10% across the five latency values. This phenomenon has a weak relationship to access patterns. For instance, applications of Type I have significant divergence: *HOT* and *2DC* follow the expected trend; however, *LEU* and *LMD* have negligible performance degradation.

To better understand the reason behind this phenomenon, we compared the cycles consumed by two important events that cause long-latency operations: cache misses and page faults. We recorded the start cycle and the end cycle of each cache miss and page fault. The "cache" refers to all caches in the GPU memory hierarchy, including L1 data cache, constant cache, texture cache, and shared L2 cache. Then we calculated the overlapped cycles among all cache misses and page faults separately. Next, we overlapped cache misses with page faults to get the cycles consumed by cache misses only and the cycles consumed by page faults only, as well as overlapped cycles. We denote total cycles consumed in the memory hierarchy as C_{total}, overlapped cycles between cache misses and page faults as $C_{overlap}$, and cycles consumed by cache misses only and page faults only as C_{cache} and C_{page}, respectively. Therefore, we can compute:

$$C_{total} = C_{cache} + C_{overlap} + C_{page} \tag{1}$$

The total cycles consumed by cache misses and page faults are C_{cache_total} and C_{page_total}, respectively. Thus,

$$C_{cache\_total} = C_{cache} + C_{overlap} \tag{2}$$

and

$$C_{page\_total} = C_{page} + C_{overlap} \tag{3}$$

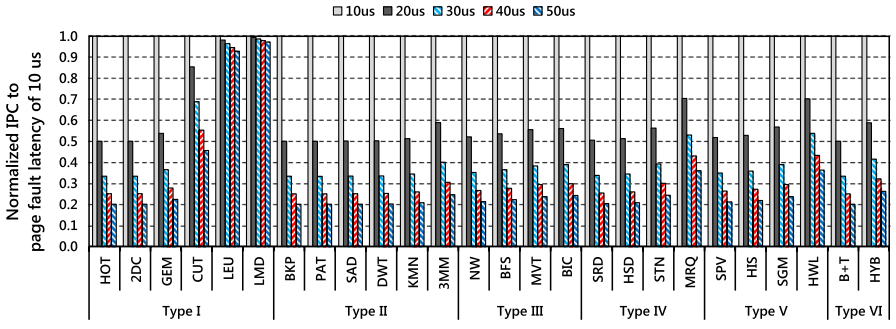


Fig. 3 Application sensitivity to page fault latency

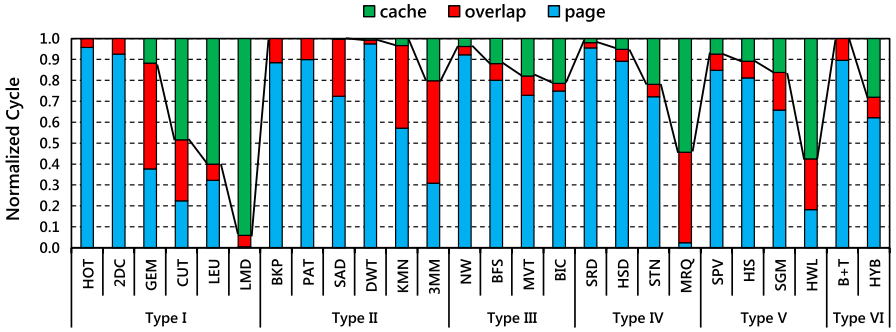


Fig. 4 Breakdown of total cycles consumed in memory hierarchy

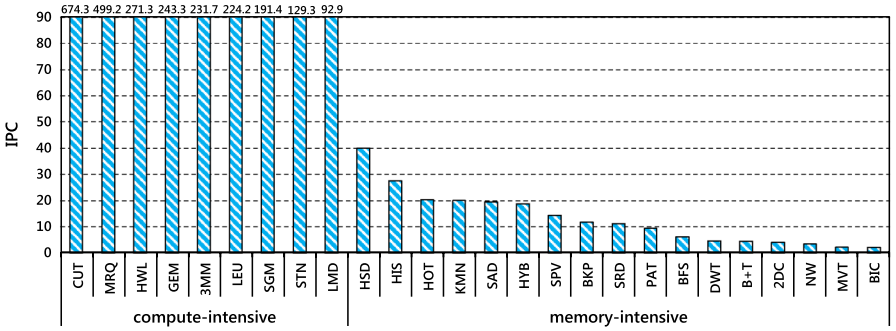


Fig. 5 IPC with page fault latency set to 10 µs

Finally, we calculated the ratios of C_{cache}, $C_{overlap}$, and C_{page} by dividing C_{total}. Figure 4 plots a breakdown of total cycles consumed in the memory hierarchy.

We also compared the IPC of the applications with a page fault latency of 10 µs. We used a simpler method (compared to prior work [16]) to classify these applications as compute-intensive and memory-intensive. If an application's IPC is larger than 90, it is compute-intensive; otherwise, it is regarded as memory-intensive. The result is shown in Fig. 5.

Figures 4 and 5 show that all page fault latency-insensitive applications are compute-intensive. More than half of their cycles are consumed by cache misses that cannot be overlapped by page faults. This is particularly true for *LMD*—more than 90% of total cycles are consumed by cache misses (L1 data cache). For *CUT* and *HWL*, the majority of cache misses are L1 data cache misses. For *MRQ*, the constant cache dominates cache misses. Yet, in *LEU*, the constant cache and texture cache cause the majority of misses. Among compute-intensive applications, *GEM*, *3MM*, *SGM*, and *STN* are more sensitive to page fault latency. They also have less percentage (less than 20%) of cycles consumed by cache misses only. Memory-intensive applications follow this trend as well and C_{page_total} (C_{page} + $C_{overlap}$) dominates the cycles consumed in memory hierarchy.

Summary. According to this analysis, we make two observations. First, most applications are sensitive to page fault latency, and hence, page faults are a performance bottleneck. Second, page fault latency-insensitive applications usually have a higher IPC (compute-intensive) and spend the majority of their cycles on cache misses that cannot be overlapped by page faults.

7 Oversubscription and eviction policy analysis

We examined how oversubscription impacts performance of LRU at three rates: 75%, 50%, and 25%. We compared LRU to three alternative eviction policies: Random, CAR, and CLOCK-Pro. All policies, except Random, are perfect (i.e., ideal), which means they track all page walk information (including page walk hits and page faults). Finally, we explore the performance gap between perfect LRU and real LRU, which only tracks page fault information.

7.1 Oversubscription analysis

We used default configurations and perfect LRU in this evaluation. Perfect LRU has precise information about page accesses. We used IPC and the number of evictions as primary metrics. The performance is normalized to a configuration with enough memory that the full application footprint fits in memory. The number of evictions is normalized to an oversubscription rate of 75%. The results are shown in Figs. 6 and 7.

The performance difference between access patterns is significant. For applications of Type I, LRU performs similarly for *HOT*, *2DC*, *LEU*, and *LMD* (in terms of evictions). As *LEU* and *LMD* are insensitive to page fault latency, they experience negligible performance degradation. Despite slightly more evictions, *CUT* has less performance degradation due to low sensitivity to page fault latency. LRU performs poorly for *GEM*, especially when the GPU memory is 25% oversubscribed. In this scenario, LRU evicts 8× more pages compared to a 75% oversubscription rate. Consequently, it incurs 82% performance loss.

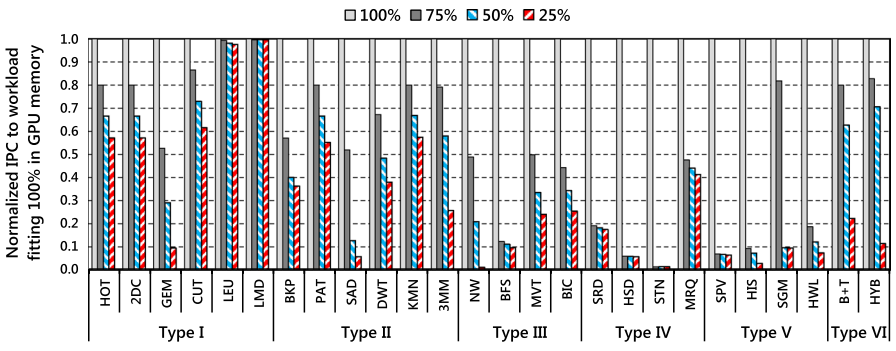


Fig. 6 Performance of LRU at 75%, 50%, and 25% oversubscription

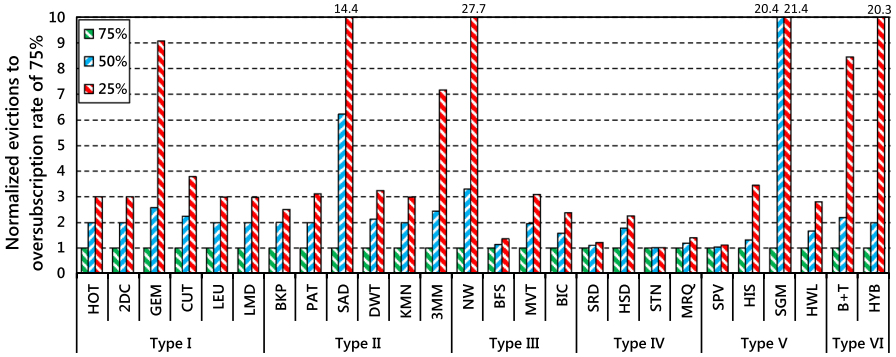
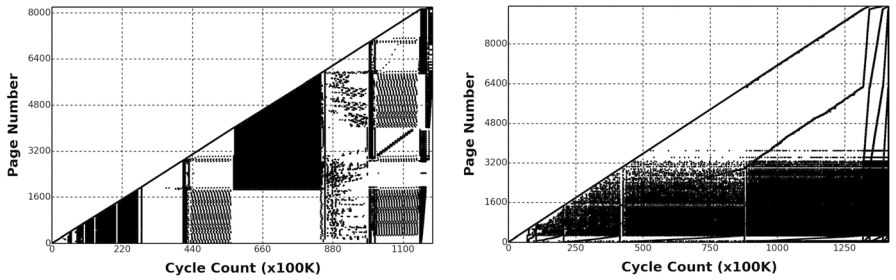


Fig. 7 Evictions of LRU at 75%, 50%, and 25% oversubscription

For applications of Type II, LRU performs similarly for *BKP*, *PAT*, *DWT*, and *KMN*, with notable exceptions of *SAD* and *3MM*. *SAD* is quite sensitive to memory capacity: When the GPU memory is 50% and 25% oversubscribed, LRU evicts 5× and 13× more pages compared to an oversubscription rate of 75%; this translates to 75% and 89% performance degradation, respectively. Compared to *SAD*, *3MM* is less sensitive to memory capacity. LRU evicts 6× more pages when the GPU memory is 25% oversubscribed.

For Type III, LRU performs similarly for *MVT* and *BIC*. However, LRU does quite differently for *NW* and *BFS*: *NW* is sensitive to memory capacity, and *BFS* is insensitive to memory capacity. When the GPU memory is 25% oversubscribed, LRU evicts 26× more pages than it does at an oversubscription rate of 75% for *NW*, which in turn incurs more than 40x performance degradation. LRU performs similarly under the three oversubscription rates for *BFS*. However, even when the GPU memory is 75% oversubscribed, the performance slowdown is significant (more than 8×). Although these applications have the same type, they have a non-negligible difference in their actual access patterns. We show the access patterns for *NW* and *BFS* in Fig. 8. *NW*'s pages are usually referenced on several different occasions. When the GPU memory is 25% oversubscribed (can hold around 2000 pages), to



(a) Access Pattern of *NW* **(b)** Access Pattern of *BFS*

Fig. 8 Access patterns of *NW* and *BFS*

make room for new pages, LRU has to evict pages that were previously referenced. However, these evicted pages are needed in the near future, and hence face severe thrashing. *BFS* has a different access pattern compared to *NW*. Only 1/3 pages are frequently referenced, with the rest of pages repeating just several times. Through further analysis, we found that at all oversubscription rates, LRU evicts these frequently referenced pages, which causes severe thrashing. Therefore, LRU performs similarly for *BFS* under different rates.

For Type IV applications, there are three observations. First, LRU performs poorly for applications with thrashing access patterns, which corroborates previous work [17, 19, 33]. For this pattern, LRU has no hits due to thrashing. Second, LRU performs similarly at each oversubscription rate. *HSD* has a slightly larger difference between the rates. Third, performance degradation is related to an application's pattern repeat. For example, *MRQ* touches its pages twice, and it has the least performance degradation. However, *STN*'s pages are touched more than 41 times. As a result, it has the most significant performance slowdown.

Type V applications (except *SGM*) have a similar trend as Type IV because both patterns are cyclic-repetitive. For *SGM*, LRU performs quite differently when the GPU memory is 75% oversubscribed and when the GPU memory is 50% or 25% oversubscribed. This is related to its access pattern: Around 75% of pages are cyclic-repetitive, and the remaining pages are simply referenced several times. When the GPU memory is 75% oversubscribed, less frequently touched pages are evicted, leaving cyclic-repetitive pages in the GPU memory. However, when the oversubscription rate is 50% or 25%, some cyclic-repetitive pages are evicted as well, which causes thrashing. Finally, LRU performs well for Type VI applications when the GPU memory capacity is larger than address region size, after which point thrashing occurs.

To study the upper bound of performance, we estimated the performance of an ideal eviction policy (which evicts the minimum number of pages). This ideal eviction policy is not feasible in practice, and we focus only on page fault latency-sensitive applications because they experience severe performance degradation. The result is shown in Fig. 9. Compared to LRU, the ideal eviction policy can significantly improve performance of some applications that are not cyclic-repetitive, such

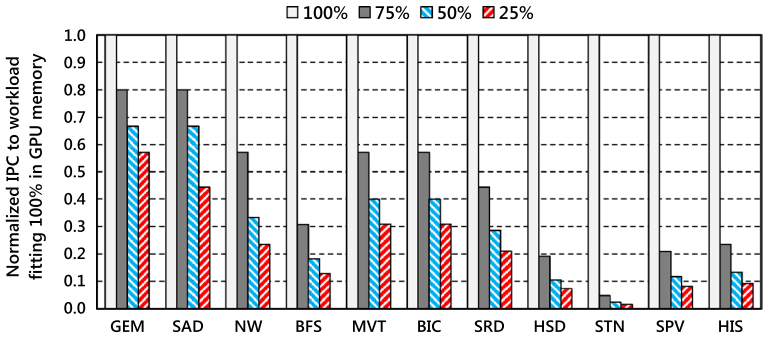


Fig. 9 Performance of an ideal eviction policy at 75%, 50%, and 25% oversubscription

as *GEM*, *SAD*, *NW*, *MVT*, and *BIC*. However, for applications with cyclic-repetitive access patterns, the improvement is limited. This is particularly true for *HSD*, *STN*, *SPV*, and *HIS*, which require other techniques to mitigate the significant performance slowdown.

Summary. On average, compared to unconstrained memory capacity, LRU achieves 0.55, 0.41, and 0.30 speedup at oversubscription rates of 75%, 50%, and 25%, respectively. Based on the analysis in this section, we make five observations. First, LRU generally performs well for applications of Types I, II, and VI. It performs poorly for applications with cyclic-repetitive access patterns (Types IV and V). Second, both eviction policy and an application's characteristics impact sensitivity to oversubscription rate. Third, when the GPU memory is significantly oversubscribed (e.g., 25%), some applications have severe performance loss. Fourth, for some cyclic-repetitive applications, the improvement by changing the eviction policy is limited, which necessities more efficient techniques to mitigate performance loss. Fifth, limited GPU memory capacity is a performance bottleneck for most applications.

7.2 Eviction policy analysis

We compared LRU to Random, CAR, and CLOCK-Pro to study how performance varies by changing eviction policy. Random is a simple policy that does not track page references. It evicts a random page in the GPU memory. CAR is a variant of the CLOCK algorithm. It addresses the disadvantages of LRU by (1) eliminating the cache hit serialization problem and it has very low overhead on cache hits, and (2) it is scan-resistant [4]. CAR is also self-tuning and requires no tunable parameters. CLOCK-Pro is another variant of the CLOCK algorithm. It uses reuse distance rather than recency to guide replacement decisions. To alleviate thrashing in cyclic-repetitive access patterns, CLOCK-Pro keeps information about historical accesses of some evicted pages. We used IPC and the number of page evictions as comparison metrics. The results are normalized to LRU.

7.2.1 LRU versus random

Figure 10 shows the performance of LRU and Random. For 25% oversubscription, we excluded *MVT* and *BIC* (Type III) due to program crashes with Random policy. Generally, Random outperforms LRU for applications of Types IV and V at 75% and 50% oversubscription. Besides, Random is better than LRU for Type III applications at 75% oversubscription. It also does better than LRU for Type VI at an oversubscription rate of 25%. Type IV and Type V are both cyclic-repetitive access patterns on which LRU suffers. Random alleviates thrashing by evicting pages randomly, which may evict some newly touched pages and preserve some of the working set in the GPU memory. Random has a speedup for *BFS* (Type III), whose part of access patterns are thrashing patterns. This contributes the major performance improvement over LRU at an oversubscription rate of 75%. For applications of Type VI, the region size is larger than the GPU memory capacity under oversubscription

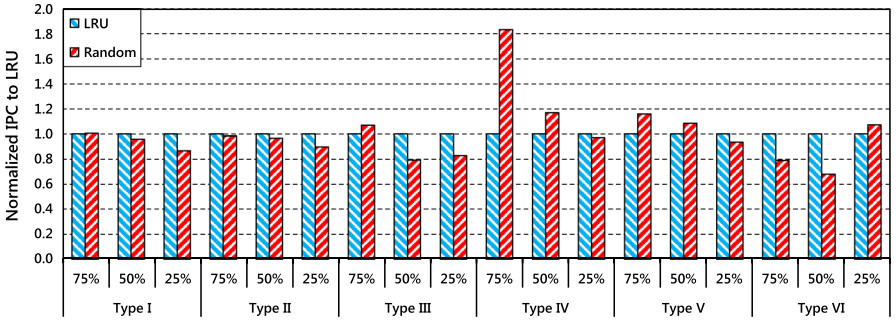


Fig. 10 Performance of LRU and Random at 75%, 50%, and 25% oversubscription

rate of 25%. In this case, LRU and Random both incur thrashing. However, Random has less thrashing. Lastly, LRU substantially outperforms Random at an oversubscription rate of 25%, except for Type VI applications. This demonstrates that when the GPU memory is highly oversubscribed, Random does poorly due to lack of page reference information, despite it outperforming LRU in some cases.

Figure 11 compares evictions of LRU and Random. As shown, Random evicts more pages in most cases except Type IV at 75% and 50% oversubscription and Type VI at 25%. Interestingly, Random evicts more pages for Type V applications when the GPU memory is 75% oversubscribed. However, it still has speedup over LRU. In this case, Random evicts 2.8× more pages for *SGM* and evicts fewer pages for the remaining applications, which increases the average evictions.

Summary. On average, Random has a 1.14×, 0.94×, and 0.93× speedup over LRU when the GPU memory is 75%, 50%, and 25% oversubscribed, respectively. Correspondingly, Random evicts 15%, 19%, and 23% more pages than LRU. On this basis, we make three observations. First, although Random outperforms LRU for cyclic-repetitive access patterns, it does worse than LRU on average. This result corroborates observations of prior work [44]. Second, when the GPU memory capacity is much smaller than the dataset size (i.e., 25% oversubscribed), Random does

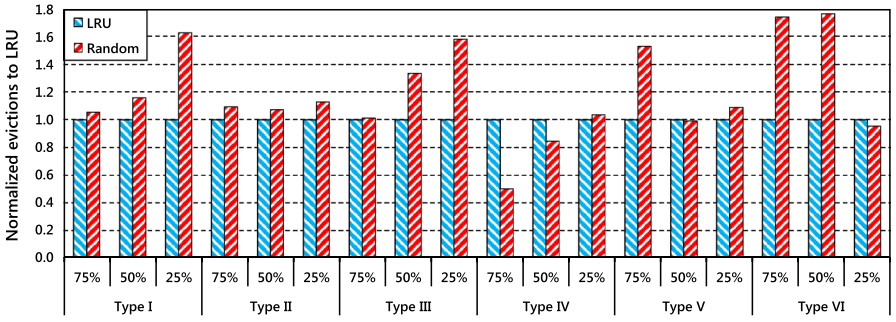


Fig. 11 Evictions of LRU and Random at 75%, 50%, and 25% oversubscription

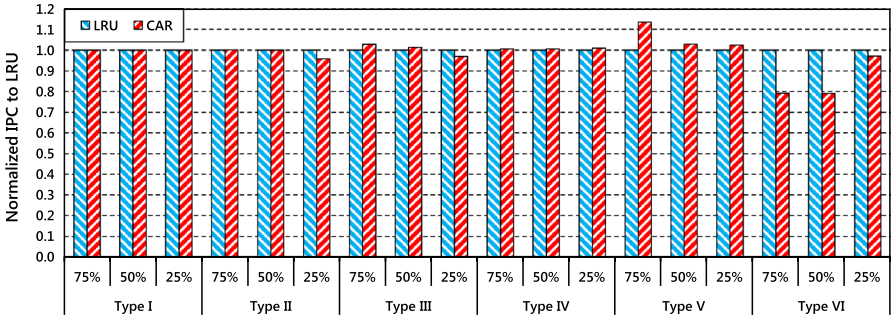


Fig. 12 Performance of LRU and CAR at 75%, 50%, and 25% oversubscription

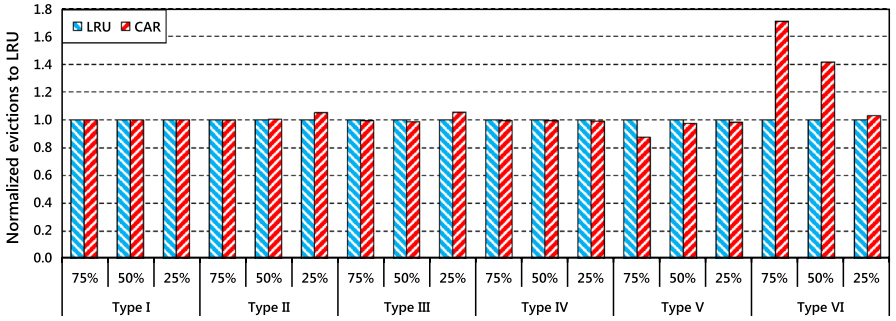


Fig. 13 Evictions of LRU and CAR at 75%, 50%, and 25% oversubscription

poorly because it lacks page reference information. Third, Random complements LRU. LRU works well for Type I and Type VI applications, but does poorly for Type IV and Type V applications. In comparison, Random works better for Types IV and V, but does poorly for Types I and VI.

7.2.2 LRU versus CAR

Figure 12 compares performance of LRU and CAR. This figure shows that CAR is similar to LRU in most cases. Notable exceptions are Type V applications at 75% oversubscription, and Type VI applications at 75% and 50% oversubscription. For Type V applications, as CAR also records frequency information, it makes better decisions by evicting pages that are referenced less frequently. However, LRU only tracks page recency information. For Type VI applications, the best policy is to evict pages in previous address regions (which will not be referenced again), just as LRU does. However, CAR considers both recency and frequency information. Hence, it evicts some pages in current regions that are needed in the future.

Figure 13 shows evictions of LRU and CAR. CAR evicts a similar number of pages as LRU in most cases, except for Type V and Type VI applications. CAR

evicts fewer pages for Type V, but evicts more than 40% of pages for Type VI at 75% and 50% oversubscription.

Summary. On average, CAR achieves 0.99×, 0.97×, and 0.99× speedup over LRU when the GPU memory is 75%, 50%, and 25% oversubscribed, respectively. Correspondingly, CAR evicts 10%, 6%, and 2% more pages than LRU. Based on the above analysis, we make three observations. First, on average, CAR performs similar to LRU. Second, CAR has the same challenge as LRU in handling thrashing access pattern (Type IV). Third, CAR performs poorly for Type VI applications.

7.2.3 LRU versus CLOCK-Pro

Figure 14 compares performance of LRU and CLOCK-Pro. This figure shows that CLOCK-Pro substantially outperforms LRU across all pattern types except for Type V at 25% oversubscription and Type VI at 75% and 50% oversubscription. CLOCK-Pro is worse than LRU for *SPV* and *SGM* when the GPU memory is 25% oversubscribed, which accounts for the major slowdown of Type V. Through further analysis, we find that the performance degradation is largely a result of maintaining the metadata for evicted pages. Using this information, CLOCK-Pro avoids thrashing by giving evicted pages a chance to become hot. However, in *SPV* and *SGM*, this design exacerbates thrashing because it keeps some pages that are less frequently in the CLOCK chain while evicting some frequently accessed pages. For Type VI, when the region fits in the GPU memory, CLOCK-Pro suffers from inefficiency despite tracking more information than LRU. In addition, compared to CAR, CLOCK-Pro has a speedup over LRU for Type IV applications because it preserves some of the working set in the GPU memory.

Figure 15 shows evictions of LRU and CLOCK-Pro. CLOCK-Pro evicts more than 40% of pages for Type V under an oversubscription rate of 25% and for Type VI under an oversubscription rate of 75% and 50%, which causes a performance loss. For Type IV applications, CLOCK-Pro evicts 40% fewer pages than LRU, which in turn translates into a 1.5× speedup. Although CLOCK-Pro evicts slightly more pages in Type I, it still has a speedup over LRU because some Type I applications are less sensitive to page fault latency, e.g., *CUT*, *LEU*, and *LMD*.

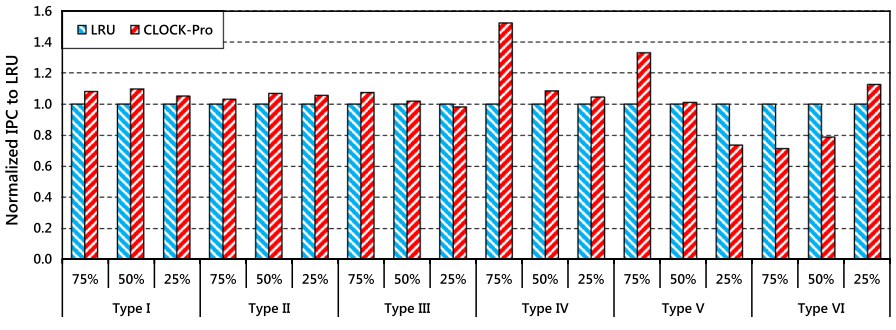


Fig. 14 Performance of LRU and CLOCK-Pro at 75%, 50%, and 25% oversubscription

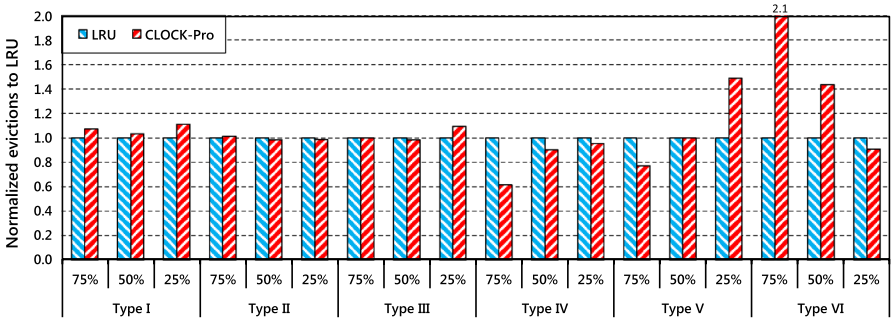


Fig. 15 Evictions of LRU and CLOCK-Pro at 75%, 50%, and 25% oversubscription

Summary. On average, CLOCK-Pro achieves 1.13×, 1.01×, and 1.00× speedup over LRU at 75%, 50%, and 25% oversubscription, respectively. Correspondingly, it evicts 10%, 6%, and 9% more pages than LRU. We make four observations. First, CLOCK-Pro achieves a small, but not significant speedup over LRU. Second, CLOCK-Pro alleviates thrashing for Type IV application. Third, CLOCK-Pro suffers from inefficiency for Type VI applications. Fourth, maintaining metadata of evicted pages does not necessarily lead to better performance.

7.2.4 Perfect LRU versus real LRU

For GPUs with unified memory, the eviction policy is managed by the GPU driver, which is invoked on page faults. Page table walks occur on the GPU. Therefore, a page eviction policy has access to only limited information since page management happens in the GPU driver (which executes on the host CPU). To examine how page walk hit information impacts the eviction policy, we evaluated the performance gap between "real" LRU and "perfect" LRU. The real LRU only tracks page fault information, and perfect LRU tracks *all* page walk information. We used default configurations at 75%, 50%, and 25% oversubscription. Performance is normalized to real LRU.

Figure 16 compares real LRU and perfect LRU when the GPU memory is 75% oversubscribed. Perfect LRU performs the same or similarly as real LRU for most applications with some outliers, e.g., *BKP*, *BFS*, *MVT*, and *SGM*. For applications of Types I, IV, and VI, the two policies have the same performance. This happens because, for Type I, all page walks are page faults, and therefore, perfect LRU and real LRU use the same information to update their chains. For Type IV, as LRU receives no hits due to failing to preserve some of the working set in the GPU memory, the two policies utilize the same information. For Type VI, although perfect LRU updates its chain with more information, once an application moves to the next address region, real LRU and perfect LRU evict the same pages when page faults occur. In other words, for Type VI, perfect LRU does not benefit from additional page walk hit information as long as the GPU memory capacity is larger than the region size.

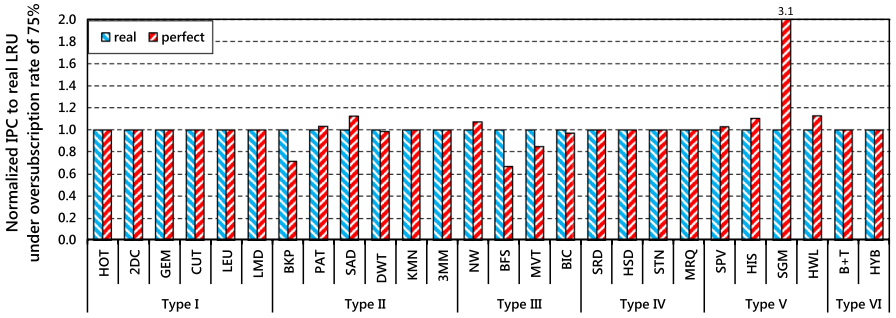


Fig. 16 Performance of real LRU and perfect LRU at 75% oversubscription

Applications of Types II, III, and V have notable performance differences due to the variance in access patterns. For Type II, perfect LRU slightly outperforms real LRU for *PAT* and *SAD*, and it is similar to real LRU for *DWT*, *KMN*, and *3MM*. However, perfect LRU is worse than real LRU for *BKP*. This is because half of this application's pages are touched again after the initial faults; real LRU evicts these pages immediately after they receive a second reference. Perfect LRU retains pages that are touched again at the MRU position, and evicts pages that have not received a second reference to make room for newly demanded pages, which incurs thrashing. Perfect LRU substantially outperforms real LRU for applications of Type V, and *SGM* has the highest speedup. In this case, perfect LRU evicts pages in *SGM* that repeat several times, which accounts for 25% of total pages. Real LRU evicts pages that are first touched and will be frequently touched in the near future. Hence, real LRU has thrashing. Most applications of Type III have performance degradation for perfect LRU. In the presence of page walk hit information, perfect LRU makes different decisions than real LRU, evicting some pages that are referenced frequently.

Figure 17 shows the result at 50% oversubscription. This case is different. First, perfect LRU performs similar to real LRU for Types I, II, and VI applications with *B+T* as an outlier. When the GPU memory is 50% oversubscribed, the address region size exceeds the GPU memory capacity and LRU has to evict some pages in current regions. In this case, perfect LRU makes better decisions. In addition,

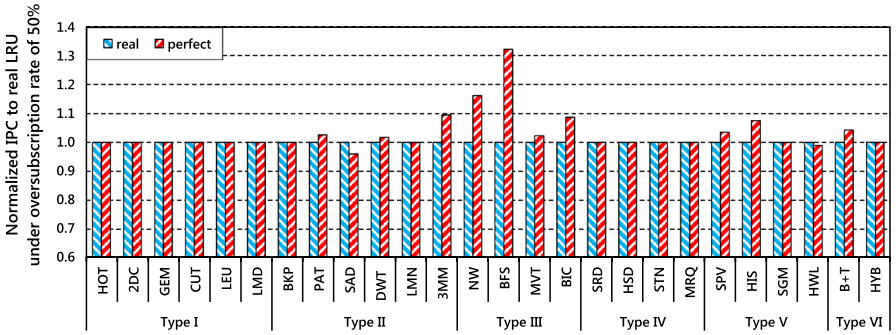


Fig. 17 Performance of real LRU and perfect LRU at 50% oversubscription

perfect LRU benefits from page walk hit information for the remaining applications, except *SAD* and *HWL*. This is particularly true for applications of Type III—perfect LRU substantially outperforms real LRU, which is quite different from the 75% oversubscription rate.

Figure 18 shows the performance of real LRU and perfect LRU when the GPU memory is 25% oversubscribed. In this situation, perfect LRU performs similarly with real LRU for most applications, except *BFS*, *MVT*, *BIC*, and *HYB*. Perfect LRU has a speedup over real LRU for the first three applications, which belong to Type III. This demonstrates that when the GPU memory is significantly oversubscribed, page walk hit information is useful to alleviate thrashing. However, in the case of *HYB*, additional hit information exacerbates thrashing, which in turn hurts performance. The overall result differs from oversubscription rates of 75% and 50%.

Summary. On average, perfect LRU only achieves 7%, 3%, and 11% performance improvement at 75%, 50%, and 25% oversubscription, respectively. We draw three observations. First, additional page walk hit information does not necessarily lead to better decisions for LRU. Second, several factors determine whether perfect LRU has a speedup over real LRU: access pattern type, the specific access pattern, and oversubscription rate. Third, real LRU works well on average, despite poor performance for some applications (e.g., *SGM*). Therefore, we conclude that there is no need to enhance LRU with additional page walk hit information.

8 Optimization opportunities

According to our analysis, long-latency page faults and memory oversubscription are two factors that cause inefficiency in GPUs with unified memory. Overcoming this inefficiency is important. This section introduces optimization opportunities that can mitigate the performance degradation due to page faults and memory oversubscription.

Page fault latency. There are two parts to page fault latency: the service time consumed in the host CPU (including CPU–GPU communication) to process an interrupt and the time consumed to transfer the demand pages or evicted pages over the

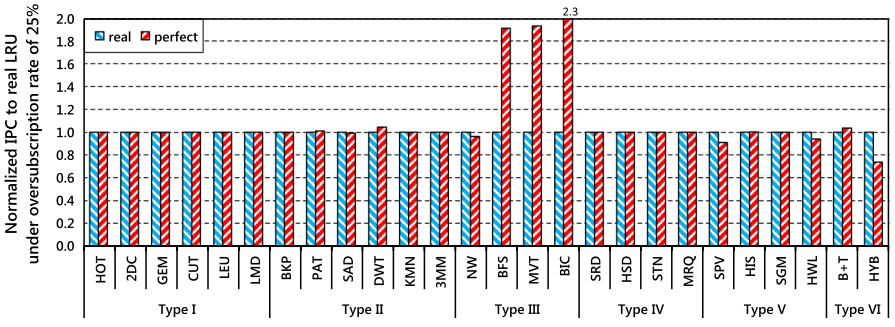


Fig. 18 Performance of real LRU and perfect LRU at 25% oversubscription

system I/O (PCIe) bus. To provide high transfer bandwidth and application scalability, NVIDIA introduced NVLink [7, 9, 29], which is a new high-speed and high-bandwidth interconnect. In NVIDIA's Volta GV100 architecture, six NVLinks provide 300 GB/s total bandwidth [30], which is 9× higher than PCIe. Using NVLink, the transfer time is much reduced. Although it is difficult to shorten service time on the host CPU, this time can be amortized by processing batches (multiple) of page faults. Prefetching and pre-eviction are two techniques that use batching. According to related work [37], the NVIDIA GPU driver prefetches a whole region of multiple continuous pages, if page accesses are dense. Zheng et al. [44] proposed *locality prefetch*, which identifies eviction candidates from the next sequential 128 virtual pages subsequent to the faulting page. This method is effective to mitigate performance degradation caused by long-latency page faults [11, 44]. Similarly, *pre-eviction* evicts multiple pages to amortize the cost of page table updates and CPU–GPU communication. Li et al. [24] proposed a similar method called *proactive eviction*, which preemptively evicts pages before the GPU runs out of memory. Two issues are with these techniques. First, page prefetch and pre-eviction are effective for applications with regular access patterns, whose behavior is easily predicted. For irregular access patterns, however, these techniques may exacerbate thrashing due to poor decisions to evict some hot pages. Second, a fixed granularity (i.e., number of pages) to prefetch and pre-evict pages [24, 44] may not work well for all cases, and hence, adapting the granularity dynamically can be useful.

Memory oversubscription. Applications with different access patterns usually react differently to memory oversubscription. For LRU-friendly access patterns, such as Type I and Type VI, page prefetch and pre-eviction can alleviate performance degradation [11, 24]. For LRU-averse access patterns, such as Type IV and Type V, an effective eviction policy is required. Yu et al. [42, 43] proposed *hierarchical page eviction (HPE)* to alleviate trashing caused by LRU for cyclic-repetitive access patterns. For this pattern, HPE selects eviction candidates from the MRU position rather than the LRU position to preserve the working set in the GPU memory. Unfortunately, even a perfect eviction policy can cause severe performance degradation for this pattern. Therefore, for this pattern and irregular access patterns, other techniques are required. Li et al. [24] proposed memory-aware throttling and capacity compression. Memory-aware throttling can reduce the working set size of an irregular application by limiting the number of concurrent threads via throttling. Capacity compression can improve the effective capacity of GPU memory. As GPU applications have a great deal of richness and variation in access patterns, a one-size-fits-all characterization does not work well. Therefore, a memory access pattern detector could be used to detect an application's pattern and select an appropriate set of techniques to get the best performance.

9 Related work

This paper (1) explored and classified access patterns (in page table walks) of GPU applications, and (2) conducted a quantitative evaluation and comprehensive analysis of unified memory, which focused on performance-related parameters, memory

oversubscription, and eviction policies. This section describes previous research focusing on evaluating performance of GPUs with unified memory and proposing techniques to address the inefficiency of unified memory.

9.1 Performance evaluation

Landaverde et al. [23] evaluated the performance of unified memory access (UMA) for NVIDIA GPUs. They found that the majority of applications and memory access patterns had significant performance overheads associated with UMA. Li et al. [25] evaluated unified memory (in CUDA 6.X) on two GPUs. They found that unified memory caused 10% performance loss on average. Dashti et al. [8] analyzed common memory management methods in CUDA and OpenCL. They found that for applications without concurrent CPU/GPU accesses to the same data, the *managed* method (used in unified memory) provided equivalent performance to default (manually managed) with substantially better programmability. Jarząbek et al. [18] evaluated performance of unified memory and dynamic parallelism for real parallel applications compared to standard CUDA API versions. They found that, in most cases, unified memory had performance degradation. Our work has two major differences with the prior research. First, the prior research focused on unified memory in early GPUs and CUDA versions (typically 6.X), which does not support demand paging. Our work evaluated the performance of GPU applications on an infrastructure that supports both unified memory and demand paging. Second, the prior research is based on real GPU hardware, where architectural parameters cannot be varied. Our study used a simulator to vary important parameters (e.g., memory capacity and page fault latency) and eviction policies (e.g., LRU, Random, CAR, and CLOCK-Pro).

Vesely et al. [41] analyzed shared virtual memory across the CPU and an integrated GPU using real measurements. They made three observations: (1) Serving a TLB miss from the GPU can be an order of magnitude slower than from the CPU; (2) divergence in memory accesses impacts GPU address translation more than the rest of the memory hierarchy; (3) page faults from the GPU are considerably slower than from the CPU. This prior work focuses on GPUs that support unified memory and demand paging. However, they targeted on integrated on-die GPUs, while we base our work on discrete PCIe-attached GPUs.

Zheng et al. [44] evaluated the impact of page prefetch and memory oversubscription using a unified memory infrastructure. They compared the performance of LRU and Random, and a study of sensitivity to oversubscription. However, the major contributions of their work were the replayable far-fault mechanism and the locality prefetcher. The authors did a simple analysis of eviction policy and oversubscription. Li et al. [24] motivated the need for memory oversubscription management by addressing oversubscription overhead in GPUs. In addition, they tested their framework's sensitivity to page fault latency. However, the major contribution of their work was not performance evaluation. Although the prior work has some discussion of performance for unified memory, the primary goal was proposing techniques to enhance GPUs with unified memory. In comparison, our work focuses

on exploring typical access patterns in GPU applications to quantitatively evaluate and analyze interaction of applications and the GPU for unified memory.

9.2 Techniques and designs

Unified memory requires address translation. Power et al. [32] and Pichai et al. [31] proposed TLB and page table walker designs for efficient address translation. Hao et al. [13] proposed similar designs for accelerator-centric architectures. To support efficient address translation for irregular GPU applications, Shin et al. [38] optimized the order of page walks by prioritizing translation requests from the instructions that require less work to service their address translation requests. They also proposed mechanisms to coalesce address translation of all pending page table walkers in the same neighborhood, which happen to have address mappings that map to the same cache line [39]. To support multiple page sizes, Ausavarungnirun et al. [1] proposed *Mosaic*, a GPU memory manager that uses base pages to transfer data over the system I/O bus, and allocates physical memory to preserve base page contiguity, ensuring that a large page frame contains pages from only a single memory protection domain. In addition, to support multi-application concurrency in GPUs, they proposed *MASK*, a new framework for low-overhead virtual memory for the concurrent execution of multiple applications [2].

Page faults cause stalls in GPUs. Zheng et al. [44] proposed replayable far-faults to enable SMs to continue execution in the presence of page faults. To mitigate the performance loss due to memory oversubscription, Li et al. [24] proposed a framework for memory oversubscription management in GPUs that incorporates: proactive eviction, memory-aware throttling, and capacity compression. These designs mentioned address the inefficiency of unified memory. Some of the research makes several similar observations with our work.

10 Conclusion

Recent support for memory virtualization in GPUs has simplified programming of applications for these architectures. However, memory virtualization also incurs overhead due to page faults and memory oversubscription. This paper describes a quantitative evaluation of unified memory. Based on our analysis, we propose optimization opportunities to mitigate performance slowdown caused by long-latency page faults and memory oversubscription, which are two key bottlenecks for unified memory. For regular applications, we find that page prefetch and pre-eviction are effective to mitigate performance loss. For applications with cyclic-repetitive access patterns, despite alleviating thrashing, an advanced page eviction policy, HPE, has limited performance improvement. For irregular applications, page prefetch and pre-eviction may even make the loss worse. Therefore, for cyclic-repetitive and irregular applications, a new approach is required for the hardware/software to cooperatively tune and select a technique based on access pattern to further alleviate thrashing.

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

References

1. Ausavarungnirun R, Landgraf J, Miller V, Ghose S, Gandhi J, Rossbach CJ, Mutlu O (2017) Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In: Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture, pp 136–150
2. Ausavarungnirun R, Miller V, Landgraf J, Ghose S, Gandhi J, Jog A, Rossbach CJ, Mutlu O (2018) MASK: redesigning the GPU memory hierarchy to support multi-application concurrency. In: Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp 503–518
3. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing CUDA workloads using a detailed GPU simulator. In: Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp 163–174
4. Bansal S, Modha DS (2004) CAR: clock with adaptive replacement. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp 187–200
5. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of 2009 IEEE International Symposium on Workload Characterization, pp 44–54
6. Choquette J, Giroux O, Foley D (2018) Volta: performance and programmability. IEEE Micro 38(2):42–52
7. Danskin J (2016) PASCAL GPU WITH NVLINK. http://hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.22-Monday-Epub/HC28.22.10-GPU-HPC-Epub/HC28.22.121-Pascal-GPU-DanskinFoley-NVIDIA-v06-6_7.pdf. Accessed 5 May 2019
8. Dashti M, Fedorova A (2017) Analyzing memory management methods on integrated CPU-GPU systems. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, pp 59–69
9. Foley D, Danskin J (2017) Ultra-performance pascal GPU and NVLink interconnect. IEEE Micro 37(2):7–17
10. Fung WW, Sham I, Yuan G, Aamodt TM (2007) Dynamic warp formation and scheduling for efficient GPU control flow. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp 407–420
11. Ganguly D, Zhang Z, Yang J, Melhem R (2019) Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In: ISCA, pp 224–235
12. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: Proceedings of 2012 Innovative Parallel Computing, pp 1–10
13. Hao Y, Fang Z, Reinman G, Cong J (2017) Supporting address translation for accelerator-centric architectures. In: Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture, pp 37–48
14. Harris M (2013) Unified memory in CUDA 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/. Accessed 8 May 2019
15. Hestness J, Keckler SW, Wood DA (2014) A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In: Proceedings of 2014 IEEE International Symposium on Workload Characterization, pp 150–160
16. Jain A, Khairy M, Rogers TG (2018) A quantitative evaluation of contemporary gpu simulation methodology. Proc ACM Meas Anal Comput Syst 2(2):35
17. Jaleel A, Theobald KB, Steely Jr SC, Emer J (2010) High performance cache replacement using re-reference interval prediction (RRIP). In: Proceedings of the 37th International Symposium on Computer Architecture, pp 60–71
18. Jarząbek Ł, Czarnul P (2017) Performance evaluation of unified memory and dynamic parallelism for selected parallel cuda applications. J Supercomput 73(12):5378–5401
19. Jiang S, Chen F, Zhang X (2005) CLOCK-Pro: an effective improvement of the CLOCK replacement. In: Proceedings of USENIX Annual Technical Conference, pp 323–336

20. Jog A, Kayiran O, Mishra AK, Kandemir MT, Mutlu O, Iyer R, Das CR (2013) Orchestrated scheduling and prefetching for GPGPUs. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, pp 332–343
21. Kehne J, Metter J, Bellosa F (2015) 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
22. Xu JY (2008) OpenCL – the open standard for parallel programming of heterogeneous systems. https://pdfs.semanticscholar.org/fb16/3d7fe546bb950294ffaf5ef6e225f630c76d.pdf. Accessed 14 Nov 2019
23. Landaverde R, Zhang T, Coskun AK, Herbordt M (2014) An investigation of unified memory access performance in CUDA. In: Proceedings of 2014 IEEE High Performance Extreme Computing Conference, pp 1–6
24. Li C, Ausavarungnirun R, Rossbach CJ, Zhang Y, Mutlu O, Guo Y, Yang J (2019) A framework for memory oversubscription management in graphics processing units. In: Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating System
25. Li W, Jin G, Cui X, See S (2015) An evaluation of unified memory technology on NVIDIA GPUs. In: Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp 1092–1098
26. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) NVIDIA tesla: a unified graphics and computing architecture. IEEE Micro 28(2):39–55
27. NVIDIA (2009) NVIDIA next generation CUDA compute architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed 10 May 2019
28. NVIDIA (2018) CUDA C programming guide. https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf. Accessed 14 Nov 2019
29. NVIDIA (2016) Pascal P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Accessed 10 May 2019
30. NVIDIA (2017) TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed 10 May 2019
31. Pichai B, Hsu L, Bhattacharjee A (2014) Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In: Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp 743–758
32. Power J, Hill MD, Wood DA (2014) Supporting x86-64 address translation for 100s of GPU lanes. In: Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture, pp 568–578
33. Qureshi MK, Jaleel A, Patt YN, Steely SC, Emer J (2007) Adaptive insertion policies for high performance caching. In: Proceedings of the 34th International Symposium on Computer Architecture, pp 381–391
34. Rogers TG, O'Connor M, Aamodt TM (2012) Cache-conscious wavefront scheduling. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp 72–83
35. Sakharnykh N (2016) Beyond GPU memory limits with unified memory on pascal. https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/. Accessed 11 May 2019
36. Sakharnykh N (2017) Unified memory on pascal and volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf. Accessed 11 May 2019
37. Sakharnykh N (2018) 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. Accessed 11 May 2019
38. Shin S, Cox G, Oskin M, Loh GH, Solihin Y, Bhattacharjee A, Basu A (2018) Scheduling page table walks for irregular GPU applications. In: Proceedings of the 45th International Symposium on Computer Architecture, pp 180–192
39. Shin S, LeBeane M, Solihin Y, Basu A (2018) Neighborhood-aware address translation for irregular GPU applications. In: Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture, pp 352–363

40. Stratton JA, Rodrigues C, Sung I, Obeid N, Chang L, Anssari N, Liu GD, Hwu WW (2012) Parboil: a revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report, pp 1–12
41. Vesely J, Basu A, Oskin M, Loh GH, Bhattacharjee A (2016) Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In: Proceedings of 2016 IEEE International Symposium on Performance Analysis of Systems and Software, pp 161–171
42. 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. https://doi.org/10.1109/TCAD.2019.2944790
43. Yu Q, Childers B, Huang L, Qian C, Wang Z (2019) Hierarchical page eviction policy for unified memory in GPUs. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software, pp 149–150
44. Zheng T, Nellans D, Zulfiqar A, Stephenson M, Keckler SW (2016) Towards high performance paged memory for GPUs. In: Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture, pp 345–357