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

Debashis Ganguly
Department of Computer Science
University of Pittsburgh
debashis@cs.pitt.edu

Rami Melhem
Department of Computer Science
University of Pittsburgh
melhem@cs.pitt.edu

Jun Yang
Electrical and Computer Engineering
University of Pittsburgh
juy9@pitt.edu

***Abstract*—Hardware support for fault-driven page migration and on-demand memory allocation along with the advancements in unified memory runtime in modern graphics processing units (GPUs) simplify the memory management in discrete CPU-GPU heterogeneous memory systems and ensure higher programmability. GPUs adopt to accelerate general purpose applications as they are now an integral part of heterogeneous computing platforms ranging from supercomputers to commodity cloud platforms. However, data-intensive applications face the challenge of device-memory oversubscription as the limited capacity of bandwidth-optimized GPU memory fails to accommodate their increasing working sets. Performance overhead under memory oversubscription comes from the thrashing of memory pages over slow CPU-GPU interconnect. Depending on the diverse computing and memory access pattern, each application demands special attention from memory management. As a result, the responsibility of effectively utilizing the plethora of memory management techniques supported by GPU programming libraries and runtime falls squarely on the application programmer. This paper presents a smart runtime that leverages the faults and page-migration information to detect underlying patterns in CPU-GPU interconnect traffic. Based on the online workload characterization, the extended unified memory runtime dynamically chooses and employs a suitable policy from a wide array of memory management strategies to address the issues with memory oversubscription. Experimental evaluation shows that this smart adaptive runtime provides 18% and 30% (geometric mean) performance improvement across all benchmarks compared to the default unified memory runtime under 125% and 150% device memory oversubscription, respectively.**

***Index Terms*—CPU-GPU heterogeneous systems, adaptive memory management, oversubscription, pattern detection**

I. Introduction

Today, GPUs are ubiquitous in systems ranging from high-performance computing (HPC) installations like Oak Ridge National Laboratory's El Capitan supercomputer to data-centres like Amazon EC2. GPUs are widely being adopted by general purpose applications like machine learning, and large-scale scientific workloads. Currently, discrete GPUs combined with CPUs via interconnection network dominate the GPU computing platforms over on-die integrated GPUs. In the classic "copy then execute" model, the onus of memory management had fallen squarely on application programmers. By automatically paging in and out of device-memory, unified memory runtime simplifies memory management and improves programmability.

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-1725657.

Despite higher programmability and ease of memory management, unified memory is still in its infancy. The limited capacity of the device-memory becomes a first-order performance bottleneck for data-intensive applications with increasing working sets. Ganguly et al [1] studied the interplay between software-prefetchers and page replacement algorithms under oversubscription. They proposed a new "pre-eviction" policy inspired by the semantics of the tree-based prefetcher to reduce page thrashing. Following the same direction, Yu et al introduced a hierarchical page eviction [2], and a co-ordinated prefetcher and eviction policy [3]. Researchers [4] have also explored bandwidth-aware page placement strategies to utilize overall system bandwidth in cache-coherent heterogeneous NUMA systems. Ganguly et al [5] proposed an adaptive unified memory runtime for page migration and pinning leveraging hardware-access counter and generic concepts like delayed migration and uncacheable access to host-pinned memory to address performance overhead of oversubscribed irregular applications.

Performance overhead under device-memory oversubscription heavily depends on the inherent application characteristics. An effective memory management strategy to address performance challenges with oversubscription requires a thorough understanding of the memory access pattern of the concerned workload. Although uncacheable zero-copy access to host-pinned memory and access counters-based delayed page migration mitigate the effect of device-memory oversubscription for data-intensive *irregular* applications, it is counter-productive for *regular* applications with dense, sequential memory access. Similarly, while LRU with $2MB$ huge-page granularity reduces write-back latency for *streaming* applications, it exacerbates page-thrashing for both *regular* and *irregular* applications. As a result, application programmers resort to extensive application profiling before choosing a suitable memory management strategy. Past researches relied on hardware-enhancements and compiler-annotations to dynamically detect memory access-patterns. Li et al [6] employed a hardware-counter to sample the number of coalesced memory accesses. Based on a thresholding scheme on this counter, they distinguish between *regular* and *irregular* applications. Data-reuse over multiple kernel launches is identified based on the compile-time analysis.

To this end, this paper introduces a smart, adaptive framework for oversubscription management by extending the exist-

978-3-9819263-5-4/DATE21/©2021 EDAA

ing unified runtime. Thus, it does not rely on any new hardware-enhancements or any programming annotations. The proposed runtime has three components - (i) an online pattern-detection module leveraging the page-migration decisions by the software prefetcher based on page-faults, (ii) an application-transparent policy engine to choose from a set of existing memory management strategies, and (iii) an augmented memory management module to dynamically employ page migration and pinning, and eviction policy best suited to reduce unwanted page-thrashing.

II. Background

This section provides the necessary background information about memory management strategies employed and proposed for CPU-GPU heterogeneous memory systems.

On-demand Page Migration in Unified Memory. Unified Memory provides a single virtual address space accessible from any processor in the system. In CUDA, `cudaMallocManaged` allows programs to allocate data that can be accessed by both host code and GPU kernels using a single shared pointer. The illusion of Unified Memory is realized by on-demand memory allocation and fault-driven page transfer. Zheng et al [7] introduced the concept of replayable far-fault. A far-fault occurs when the addressed memory page is not physically present in the device memory, unlike a near-fault which occurs on a cache miss. GPU memory management unit (GMMU) can only perform a page table lookup. It can not handle far-faults and update/add page table entry. The far-fault is registered in the Far-fault Miss Status Handling Registers (MSHRs). The offending warps are stalled. GPU interrupts the unified memory runtime hosted in the operating system to relay the far-faults. Based on the groups of far-faults, the software prefetcher in the runtime determines and schedules direct memory access (DMA) requests to transfer a chunk of memory from the host to the device over CPU-GPU interconnect. Corresponding pages are allocated on-demand, and page table and TLB entries are created/updated upon completion of the scheduled transfer. The MSHRs are consulted to notify the corresponding load/store unit and the memory access is replayed and the stalled warps are marked executable.

Software Prefetchers. In unified memory, GPU's massive thread-level parallelism (TLP) is not sufficient to mask far-fault handling latency. Software prefetchers have been proposed by researchers [7], [8] to relieve programmers from the burden of writing hand-tuned code using complicated `cudaMemPrefetchAsync` constructs. By studying the MIT-licensed opensource `nvidia-uvm` module and executing micro-benchmarks, Ganguly et al [1] discovered that CUDA unified memory runtime implements a tree-based software prefetcher. The user-requested size of a `cudaMallocManaged` allocation is first rounded up to the next $2^i * 64KB$. Then, the allocated size is logically divided into $2MB$ huge-pages plus a fraction of $2MB$. Each of these chunks is further logically divided into $64KB$ basic blocks, which is the unit of prefetching. Then, a set of full-binary trees are created where the leaf-levels hold $64KB$ basic-blocks and root nodes correspond to $2MB$ huge-page address. The tree-base software prefetcher is a heuristic governed by these full-binary trees. Upon receiving a set of far-faults from the GPU, the runtime calculates the base address of the $64KB$ basic-blocks corresponding to these faulty pages and identifies the corresponding leaf-nodes of the full-binary tree structures. The driver communicates the address of these basic blocks to the I/O root complex to schedule data transfers. As the blocks are migrated from the host to the device and pages are allocated, the runtime keeps track of the current occupancy or the total size of valid memory for each node in these binary trees. At any given instance, if the runtime determines that occupancy of a non-leaf node in the tree data-structures is more than 50% of the total capacity of that node, it selects non-valid leaf nodes under the current node as further prefetch candidates.

Page Eviction. Before scheduling any new page-migration from the host to the device memory, the runtime evaluates the current occupancy of device-memory by querying low-level system APIs. When there is no free space in the device memory, the runtime invokes a page replacement routine to write-back pages automatically from the device to the host memory. CUDA runtime implements Least Recently Used (LRU) page eviction with the strict granularity of $2MB$ x86-OS huge-page. Ganguly et al [1] introduced a tree-based pre-eviction inspired by the tree-based prefetcher which leverages the full-binary tree structures created and maintained by the runtime for the managed allocations. It employs a thresholding-based heuristic inverse of the prefetcher. At any instance, if the current occupancy of a non-leaf node falls below 50%, the runtime pre-evicts other $64KB$ valid leaf-nodes under it.

Page Migration and Pinning. Researchers [4], [8] have studied the trade-offs between lower-latency, uncacheable direct memory access to the host-pinned memory and bandwidth-optimized local memory access in the context of heterogeneous systems. They proposed a static page placement strategy, which requires compiler support for profiling and programmer intervention to annotate the memory allocations. NVIDIA CUDA runtime also supports placement of memory pages to either host or device memory based on user-provided hints. The `cudaMemAdviseSetAccessedBy` flag allows the device to establish direct mapping to the host memory. Further, the `cudaMemAdviseSetPreferredLocation` allows specifying the preferred location of a memory allocation to be set to the host memory. If an allocation is advised to be soft-pinned to the host memory, then the pages are not copied directly to the device at the first touch. Rather, the migration is delayed till the number of read-requests crosses a certain static threshold configured in the model-specific register. Whereas, on write access, the data is migrated irrespective of the threshold to maintain the exclusivity of access. Ganguly et al [5] implemented a dynamic threshold-based delayed migration on the access frequency and eviction count to adaptively migrate hot pages to the device-memory and pin cold pages to the host memory. It balances between the low-latency remote access and the high-bandwidth local access to reduce thrashing of memory pages for data-intensive irregular applications.

III. Motivation

This section motivates the proposed smart runtime by justifying that the CPU-GPU interconnect traffic is the source of performance bottleneck under device-memory oversubscription and highlighting the limitations of the existing research works to address this challenge.

A. Slow CPU-GPU Interconnect

Figure 1 shows that the normalized delay suffered by CPU-GPU interconnect traffic is almost twice when the available interconnect bandwidth is halved which justifies the average (geometric mean) 1.29× performance slowdown across the set of general-purpose applications. These workloads are described in Section V.

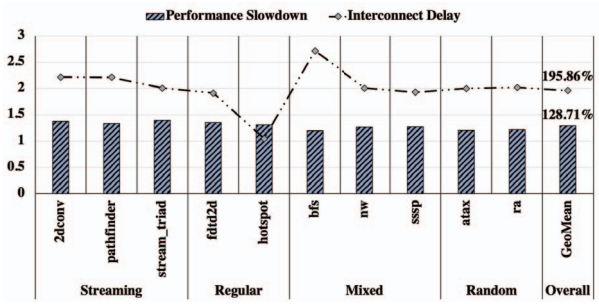


Fig. 1: The effect of halving the available interconnect bandwidth on the performance of unified memory applications. Results are obtained by executing the applications on the simulation framework described in Section V.

In contrast to the classic "copy-then-execute" model, with unified memory, kernel execution stalls for fault-driven migrations. The end-to-end execution time of an application with unified memory allocations resembles the summation of serial memory copy and kernel execution time. Even, GPU's heavy multi-threading is rendered ineffective in hiding long latency host-to-device page migrations. Hence, the CPU-GPU interconnect traffic becomes the source of performance bottleneck.

B. Oversubscription Overhead

Figure 2 shows the performance degradation of GPGPU workloads with varied percentage of memory oversubscription. The results are obtained by running the workloads on `GeForceGTX 1080 ti` *(not on simulated environment)*. To emulate memory oversubscription, working sets of the workloads are not scaled, rather the total free space is controlled by allocating dummy `cudaMalloc`ed variables because `cudaMalloc`ed allocations are pinned and do not participate in eviction.

The memory access pattern of a GPU application depends on the fine-grained parallelism and inherent memory access characteristics. Different memory-access patterns can react differently to memory oversubscription resulting in highly varied performance overheads. While for *streaming* and *regular* applications, the performance slowdown under oversubscription

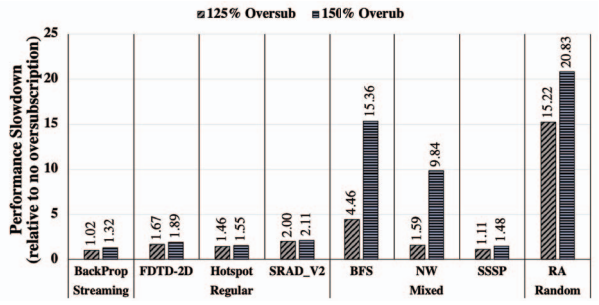


Fig. 2: Sensitivity of workloads to the percentage of memory oversubscription (performed on real hardware).

stems from fault-based migrations waiting behind long latency write-backs; for *mixed* and *random* applications, the performance overhead is due to the excessive page thrashing.

C. Limitation of Existing Works

Past researches have mainly focused on identifying device-memory access pattern either by intrusive user-based profiling or by proposing hardware extensions. For example, Yu el al [2] profiled workload execution on the simulation platform to identify six representative classes of device-memory access pattern for various general-purpose applications. They further proposed hierarchical page eviction that relies on additional micro-architectural enhancements to reduce page-thrashing. Li et al [6] also classified GPU applications in three major categories - 1) regular applications without data sharing, 2) regular applications with data sharing and 3) irregular applications. They proposed a hardware-counter in each SM's load/store unit to sample the number of coalesced memory accesses and determine the memory access pattern of the executing workload. Upon detecting the memory access pattern, their proposed runtime chooses between proactive eviction, memory-aware throttling, and capacity compression to address the challenges with memory oversubscription.

Unlike the above works, in this paper, we focus on the underlying patterns over CPU-GPU interconnect traffic. Page-migration patterns are inherently different from memory access patterns. For example, a particular workload can migrate the pages in the device memory during the cold-start and access it locally across different kernel launches over multiple iterations. If there is no oversubscription and subsequent page thrashing, the pattern of accessing device-memory has little to no effect on page thrashing and application performance after the cold-start. Page migration is a compounded behaviour of - (i) memory requirements of scheduled warps, (ii) efficacy of coalescing unit and the capacity of miss-status handling registers (MSHRs), and (iii) most importantly the heuristic employed by the software prefetcher. Under oversubscription, access pattern and eviction heuristic also indirectly influence the page-migration decision and in turn the amount of interconnect traffic. While device memory access pattern affects cache hit rate and local memory bandwidth, for applications with unified memory allocations, the page migration pattern plays a far more important role.

Unlike previous works, this paper does neither rely on compiler-annotations based on intrusive profiling nor on any hardware extension to detect an underlying pattern in CPU-GPU interconnect traffic. Rather, the pattern-detection engine, which is implemented as part of the unified memory runtime, leverages the information of $64KB$ basic-blocks identified by the software-prefetcher for fault-driven migration. Then, based on the detection result, the smart framework adaptively chooses and employs the best-suited memory management policy to reduce interconnect memory-traffic induced by page-thrashing. Hence, the proposed runtime is simple and pragmatic with no need for any programmer-assistance or new hardware enhancements.

IV. The Adaptive Unified Framework

This section describes the proposed adaptive framework for oversubscription management in CPU-GPU unified memory.

A. Memory Migration Pattern

This paper considers the following page migration patterns for the runtime's new pattern detection engine.

Regular. Equation 1 represents a sequential migration of k basic-blocks cyclically repeated over N iteration. In this set p_1 and p_k are respectively *distant* and *near* re-referenced blocks. When k is larger than the device-memory capacity, a percentage of the distant re-referenced blocks are written back to accommodate newer migrations. As a result, there is constant cyclical thrashing of memory pages.

$$(p_1, p_2, ..., p_k)^N \text{ where, } N > 1, k > \text{Memory Size} \quad (1)$$

Streaming. A streaming migration pattern is a special case of the *Regular* access pattern where no data is re-referenced. With $N = 1$, Equation 1 is transformed into Equation 2, which represents a streaming migration pattern. This pattern has no locality in its references. The memory pages have an infinite re-reference interval.

$$(p_1, p_2, ..., p_k) \text{ where, } k = \text{Number of Accesses} \quad (2)$$

Random. Equation 3 defines a *Random* migration pattern where basic-blocks, ranging between $[1, m]$, has a migration probability of ε. Further, ε is high, close to 1 for the common case. This means that memory chunks are migrated randomly based on some probability distribution.

$$P_\varepsilon(p_1, p_2, ..., p_m) \text{ where, } m = \text{Number of Accesses} \quad (3)$$

Irregular/Mixed. Equation 4 represents a *Irregular* or *Mixed* migration pattern. An *irregular* pattern can be a mix of *regular* and *random* accesses over more than one memory allocations. Equation 4 shows that basic blocks q_i, ranging between $[1, k]$, is linearly migrated for M iterations and blocks p_i, ranging between $[1, m]$, are randomly migrated with probability, ε. This entire access can be repeated for N iterations. Note that unlike *Random* migration pattern, the probability of migration, ε is almost close to 0 in the common case.

$$((q_1, q_2, ..., q_k)^M P_\varepsilon(p_1, p_2, ..., p_m))^N \quad (4)$$

B. Pattern Detection

The main contribution of this paper is the user-agnostic, pattern-detection engine augmented to the smart adaptive runtime. The heuristic employed by the pattern detection engine is illustrated by the following steps.

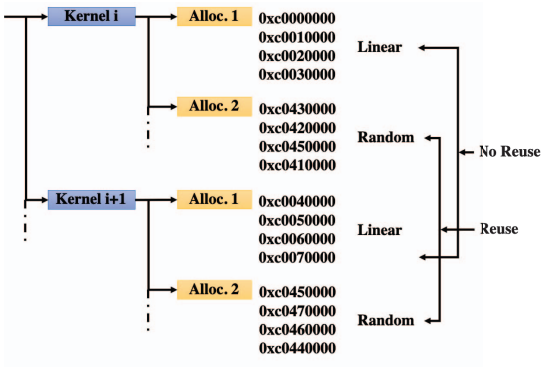


Fig. 3: An example of the hierarchical data-structure keeping track of block migration addresses used by detection engine.

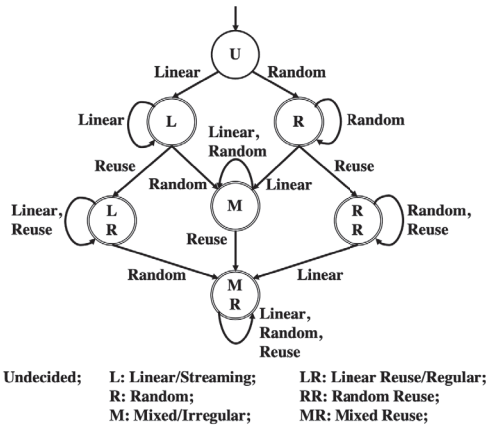


Fig. 4: Deterministic Finite Automaton (DFA) for managed allocations demonstrating the transition of migration states.

❶ GPU interrupts runtime in the host to relay the far-fault information [7]. ❷ Based on the group of faults, the software prefetcher, in unified memory runtime, determines the $64KB$ basic blocks as the migration candidates [1]. ❸ These basic blocks are communicated to I/O root complex to schedule DMA transfers. ❹ The pattern detection module of the extended runtime leverages this information of basic-block migration. Over the course of computation and data-migration, the runtime keeps a list of the basic-block addresses with their corresponding scheduling timestamp. ❺ The runtime is already aware of the base address and allocation size of the managed allocations along with the kernel launch and completion time.

❻ The detection engine is triggered at oversubscription before evoking page eviction routine. ❼ The detection module first scans through the list of basic-block transfers and segregates them at kernel boundaries based on their scheduling timestamp. Then, within each kernel boundaries, migrated blocks are further subdivided into groups of managed allocations based on their virtual address. Figure 3 shows an example of the hierarchical data-structure, used for tracking the basic-block addresses grouped by the managed allocation in each kernel boundary. ❽ Then, the detection module scans through the list of addresses in each managed allocation within kernel boundary to determine whether they show linearity/randomness of migration. ❾ Across the kernel boundaries, addresses are compared to determine any re-referencing. ❿ Based on intra-kernel pattern detection in $Step8$ and inter-kernel detection of re-referencing $Step9$, detection module refines the state of managed allocations individually. Figure 4 shows the possible state transitions starting from the initial *Undecided* state.

C. Adaptive Memory Management

The proposed runtime is the culmination of the lessons learned from the past researches on unified memory oversubscription [1], [5]. Based on the verdict of the pattern-detection module, the policy-engine chooses from the following set of memory management strategies and employs it dynamically. As computation progresses and the detection engine perfects its prediction, the runtime is capable of adaptively switching between memory management techniques to cater to the current prediction at its best. *Note that the runtime starts with the tree-based prefetcher and huge-page LRU eviction as the base strategy and based on the detected patterns, adapts its policies as described below:*

Regular/Linear with data-reuse. LRU always attempts to evict the distant re-referenced block p_1 which is accessed in the immediate next cycle. As a result, LRU causes cyclical thrashing of memory pages. The situation is further worsened by huge-page ($2MB$) eviction granularity. The goal of a good eviction algorithm is to avoid eviction as much as possible and also seamlessly coordinate with the software prefetcher. Ganguly et al [1] showed the combination of tree-based prefetcher and tree-based pre-eviction can reduce page thrashing for allocations with both regular and irregular migration pattern.

Streaming/Linear with no data-reuse. As there is no data reuse and a large array is scanned linearly, the goal for memory management here is to replace memory pages at the highest granularity of $2MB$ huge-pages. This is because the performance overhead of device-memory oversubscription for streaming pattern stems from the write-back latency of evicted blocks. Larger eviction granularity ensures lower write-back latency. Hence, for streaming migration pattern, smart runtime sticks with the default eviction policy of huge-page LRU.

Random (with or without data-reuse). As memory is migrated randomly, the prefetcher is not effective to coalesce multiple transfers into a single larger unit. The ideal memory management strategy here is to hard pin the oversubscribed portion of managed allocation in the host memory and allow device access the memory at sub-page granularity (up to $128B$) sporadically over interconnect without either caching the bytes or migrating the pages to the device.

Mixed (with or without data-reuse). From the Equation 4, memory pages (q_i) with linear, sequential pattern with re-referencing can be classified as *hot* pages and the allocation with random, sparse access (p_i) as *cold*. *hot* and *cold* pages are pinned on the device and the host memory respectively. This is achieved by employing the adaptive page pinning and delayed migration heuristic proposed by Ganguly et al [5]. However, tree-based pre-eviction is employed instead of access counter-based LFU with $2MB$ granularity to reduce page-thrashing.

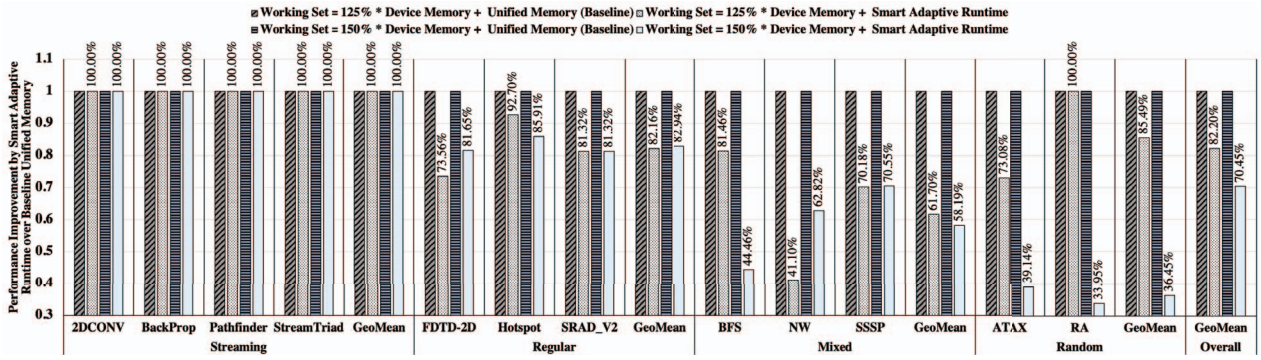
V. Experimental Methodology

Simulation Framework. Ganguly et al [1], [5] extended GPGPU-Sim 3.x [9] to provide functional and timing simulation support for Unified Memory such that benchmarks written using `cudaMallocManaged`, and `cudaDeviceSynchronize` APIs can be simulated. We extended GPGPU-Sim UVM Smart [10] by adding - (i) the detection engine to identify the pattern in CPU-GPU interconnect traffic, (ii) a dynamic policy engine that chooses from a wide array of existing memory management policy, and (iii) an augmented memory management module that adaptively switches between policies.

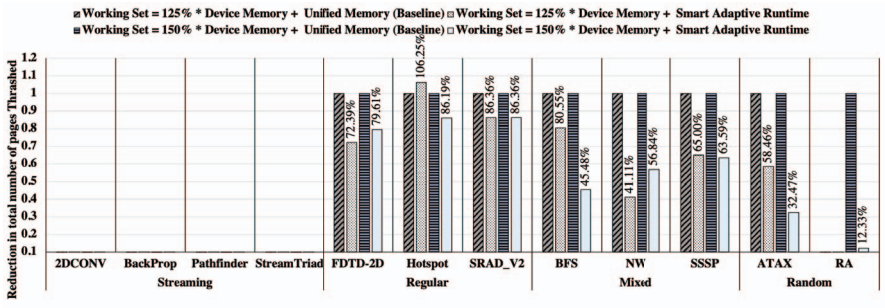
Application Suite. Along with the simulation framework, GPGPU-Sim UVM Smart [10] also provides a set of applications from Rodinia [11], Lonestar [12], and PolyBench [13] benchmark suites. These are modified to use CUDA Unified Memory APIs. As discussed in Section IV-A, we divide these benchmarks into four categories: (i) streaming (`2dconv`, `backprop`, `pathfinner`, `stream_triad`), (ii) regular (`fdtd-2d`, `hotspot`, `srad_v2`), (iii) mixed (`bfs`, `nw`, `sssp`), and (iv) random (`atax`, `ra`). To simulate oversubscription, working sets of the benchmarks are not scaled, rather the available free space in the device-memory is controlled by a configuration parameter of the simulation setup.

VI. Evaluation

To evaluate the extended framework, we compare the execution time of benchmark kernels running with the proposed smart adaptive framework against the execution time of kernels running with the NVIDIA's default unified memory runtime. Figure 5a shows that the extended framework provides an average (geometric mean) 28% and 30% performance improvement under 125% and 150% memory over-subscription respectively. Note that for applications with streaming migartion pattern, the performance or smart adaptive runtime is the same as default unified memory. This is because the memory management strategies are the same for the two as described in Section IV-C. Whereas, for applications with other migration patterns, smart runtime is extremely effective compared to the default unified memory. By dynamically determining an adaptive memory management strategy, the smart runtime ensures considerable reduction in page thrashing as shown in Figure 5b which contributes to the performance speed-up. Note as mentioned earlier, Figure 5b shows no page thrashing for streaming applications due to the zero-reuse of data.



(a) Performance Speedup



(b) Reduction in Thrashing

Fig. 5: Performance of smart adaptive framework compared to unified runtime

VII. Conclusion

This paper introduces an application-aware adaptive framework, that extends the existing software-managed runtime without requiring any new hardware enhancements. To the best of our knowledge, this is the first work to propose a runtime-based detection engine to identify the underlying pattern in CPU-GPU interconnect traffic of general-purpose GPU applications with unified memory. The smart runtime adapts to the application characteristics and dynamically employs the best-suited memory management strategy to reduce thrashing of memory-pages under device-memory oversubscription.

References

[1] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 224–235.

[2] 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*, 2019.

[3] Q. Yu, B. Childers, L. Huang, C. Qian, H. Guo, and Z. Wang, "Coordinated page prefetch and eviction for memory oversubscription management in gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 472–482.

[4] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.

[5] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 451–461.

[6] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 49–63.

[7] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 345–357.

[8] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 354–365.

[9] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[10] Debashis Ganguly, "GPGPU-Sim UVM Smart," https://github.com/DebashisGanguly/gpgpu-sim_UVMSmart/, 2018.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[12] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*. Ieee, 2012, pp. 1–10.