# Adaptive Memory Management for CPU-GPU Heterogeneous Systems

by

**Debashis Ganguly**

Bachelor of Technology in

Computer Science and Engineering, West Bengal University of Technology, 2009

Submitted to the Graduate Faculty of

the Department of Computer Science, School of Computing and Information

in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy**

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Debashis Ganguly

It was defended on

October 13, 2020

and approved by

Rami Melhem, Department of Computer Science

Jun Yang, Electrical and Computer Engineering Department

Youtao Zhang, Department of Computer Science

Bruce Childers, Department of Computer Science

Dissertation Director: Rami Melhem, Department of Computer Science

To maa and baba,
Jyotsna and Malay Ganguly,
who mean the world to me

**Acknowledgements**

First and foremost, my heartfelt gratitude goes to my advisor, Dr. Rami Melhem for his guidance and patience throughout my time in graduate school. He taught me the fundamentals of architecture during my early coursework, and in my later years taught me about the art of effective communication. I also want to thank my co-advisor, Dr. Jun Yang. They have always been available to discuss research ideas and have spent countless hours reviewing and improving my work. I would like to thank my Ph.D. committee members for their feedback and constructive criticism which greatly improved the quality of this dissertation.

I am fortunate to meet several extraordinarily kind and brilliant people during my time in graduate school. My earnest acknowledgement goes to Ziyu Zhang, who had been a pivotal contributor to the works contained in this dissertation. I would especially like to thank Michael LeBeane, who kindly mentored me during my internship at AMD. He has been kind enough to clarify my doubts and help me develop a pragmatic and technical outlook towards research. Finally, I would like to thank Prof. Daniel Mossé for his unwavering encouragement and motivation through stressful times.

Last but no way the least, my sincere gratitude goes to my parents, and my family for all their mental support and an infinite endurance with which they put up with me during this arduous phase of my life. It is really hard to paraphrase my acknowledgement to them, as it will invariably look very little, compared to their so many contributions in my life.

**Adaptive Memory Management for CPU-GPU Heterogeneous Systems**

Debashis Ganguly, PhD

University of Pittsburgh, 2020

High compute-density with massive thread-level parallelism of Graphics Processing Units (GPUs) is behind their unprecedented adoption in systems ranging from data-centers to high-performance computing installations. Currently, discrete GPU(s) combined with CPU via slow CPU-GPU interconnect dominate these computing platforms. The introduction of on-demand paging and fault-driven migration support in the newer generation GPUs, powered by software-managed unified memory runtime, simplified memory management in the CPU-GPU heterogeneous memory systems and ensured higher programmability. As GPUs are increasingly being used to accelerate general-purpose applications beyond traditional graphics processing, these systems raise a number of design challenges, including smart runtime systems, programming libraries, and micro-architecture.

One of the key challenges this dissertation aims to address is the performance slowdown under device memory oversubscription. When the working set of an application exceeds the device's memory capacity, CPU-GPU interconnect-traffic from page eviction and software prefetching becomes a major source of performance bottleneck. Firstly, this dissertation proposes a pre-eviction policy, that adapts the semantics of software prefetcher to reduce the CPU-GPU interconnect traffic from unnecessary page thrashing. Secondly, this dissertation proposes an adaptive page migration and pinning strategy for the runtime that adapts to the irregularity in the access pattern based on the frequency of memory access. Disparate applications demand special attention for memory management based on their workload characteristics, thread-level parallelism, and memory access pattern. Finally, this dissertation introduces a smart runtime that transparently caters to different classes of applications by unifying a wide array of memory management strategies. As GPUs are becoming an integral part of commodity computing clusters, assuring system throughput and execution fairness is becoming a critical challenge for multi-tenant workloads. To this end, the dissertation proposes a CPU-GPU interconnect scheduler that provisions network traffic adapting to the

disparate computation characteristics and bandwidth demands of participating applications in the composed workload. By introducing all these techniques, the dissertation makes significant progress towards realizing the goal of developing an adaptive, smart software-managed runtime for CPU-GPU heterogeneous memory systems.

## Table of Contents


**1.0 Introduction** . . . . . 1

1.1 Problem Description . . . . . 2

1.2 Thesis Statement . . . . . 8

1.3 Contributions . . . . . 9

1.4 Organization . . . . . 11

**2.0 Background and Related Work** . . . . . 12

2.1 Baseline Architecture . . . . . 12

2.2 Unified Memory . . . . . 12

2.2.1 Fault-driven Migration and On-demand Allocation . . . . . 13

2.2.2 Tree-based Software ($TBN_p$) Prefetcher . . . . . 14

2.2.3 Page Replacement . . . . . 18

2.2.4 Remote Zero-copy Access and Delayed Migration . . . . . 18

2.3 GPU Multi-tenancy . . . . . 20

2.3.1 GPU Sharing Mechanism . . . . . 20

2.3.2 CPU-GPU Interconnect Provisioning . . . . . 21

2.4 Related Work . . . . . 22

2.4.1 Prefetching and Page Replacement in Unified Memory . . . . . 22

2.4.2 Page Migration and Pinning . . . . . 22

2.4.3 Unified Framework . . . . . 24

2.4.4 Execution of Concurrent Applications on GPU . . . . . 25

2.4.5 GPU Memory Scheduling . . . . . 26

2.4.6 GPU Simulators . . . . . 27

2.4.7 GPGPU Workloads . . . . . 27

**3.0 UVMSmart: Simulation Framework and Unified Memory Benchmarks** 29

3.1 UVMSmart: Design and Implementation . . . . . 29

3.1.1 Design Requirements . . . . . 30

3.1.2 Micro-architecture Modeling . . . . . 30
3.1.3 Timing Model . . . . . 31
3.1.4 Runtime Modeling . . . . . 33
3.1.5 Unified Memory API Modeling . . . . . 36
3.1.6 Support for Concurrent Execution . . . . . 36
3.2 Application Suite . . . . . 37
3.2.1 Micro-benchmarks . . . . . 37
3.2.2 Unified Memory Benchmarks . . . . . 38
3.2.3 Unified Memory Concurrent Application Framework . . . . . 39
3.3 Configuration and Validation . . . . . 40
3.3.1 Simulation Configuration . . . . . 40
3.3.2 Containerization . . . . . 40
3.3.3 Validation . . . . . 41
**4.0 Adaptive Page Replacement** . . . . . 43
4.1 Pre-eviction Policies Adaptive to Prefetchers . . . . . 43
4.1.1 Sequential-local ($SL_e$) Pre-eviction . . . . . 43
4.1.2 Tree-based Neighborhood ($TBN_e$) Pre-eviction . . . . . 44
4.1.3 Specific Design Choices . . . . . 45
4.2 Experimental Evaluation . . . . . 47
4.2.1 Pre-eviction Policies in Isolation . . . . . 47
4.2.2 Combinations of Pre-eviction Policy and Software Prefetcher . . . . . 48
4.2.3 Memory Over-subscription Sensitivity . . . . . 50
4.2.4 Reserving Percentage of LRU Page List from Eviction . . . . . 51
4.2.5 2MB Large Page Eviction . . . . . 52
4.3 Conclusions . . . . . 53
**5.0 Adaptive Page Migration and Pinning** . . . . . 55
5.1 Motivation . . . . . 55
5.1.1 Workload Characterization . . . . . 56
5.1.2 High-level Observations . . . . . 58
5.2 Dynamic Access counter Threshold Based Delayed Data Migration . . . . . 58

5.2.1 Dynamic Access Counter Threshold . . . 59
5.2.2 Access Counter Based Page Replacement . . . 60
5.2.3 Implementation Details . . . 61
5.3 Experimental Evaluation . . . 62
5.3.1 Sensitivity to Static Migration Threshold . . . 62
5.3.2 The Case of No Oversubscription . . . 63
5.3.3 The Case of Oversubscription . . . 64
5.3.4 Sensitivity to Multiplicative Penalty . . . 66
5.3.5 Access Counter Based Eviction . . . 68
5.3.6 Invalidating Clean Pages . . . 68
5.4 Conclusion . . . 70
**6.0 An Adaptive Unified Framework** . . . 71
6.1 Motivation . . . 71
6.2 The Unified Framework . . . 72
6.2.1 Memory Migration Pattern . . . 72
6.2.2 Pattern Detection . . . 74
6.2.3 Adaptive Memory Management . . . 76
6.3 Experimental Evaluation . . . 78
6.4 Conclusion . . . 78
**7.0 Adaptive Interconnect Provisioning for Multi-tenant Workloads** . . . 79
7.1 Motivation . . . 79
7.1.1 Workload Characterization . . . 80
7.1.2 Application Interference . . . 81
7.1.3 Limitations of Existing Scheduling Schemes . . . 82
7.1.4 Interference under Device Memory Oversubscription . . . 84
7.1.5 Key Observations . . . 84
7.2 Application-aware CPU-GPU Interconnect Provisioning . . . 86
7.2.1 Performance Model . . . 86
7.2.2 Mechanism and Implementation Details . . . 88
7.3 Experimental Evaluation . . . 90

7.3.1 Evaluation Metrics . . . . . 90
7.3.2 Effect on System Throughput . . . . . 91
7.3.3 Effect on Fairness . . . . . 93
7.3.4 The Case of Oversubscription . . . . . 94
7.4 Conclusion . . . . . 97
**8.0 Concluding Remarks** . . . . . 98
8.1 Summary . . . . . 98
8.2 Future Direction . . . . . 99
**Bibliography** . . . . . 100

## List of Tables

1 Full-benchmarks with their memory access pattern. . . . . 39

2 Configuration parameters of the simulated system. . . . . 41

## List of Figures


1 Sensitivity of workloads to the percentage of memory oversubscription (performed on real hardware). . . . . 4

2 The effect of computation resources and interconnect bandwidth on unified memory applications. . . . . 7

3 Overview of the thesis work. . . . . 11

4 Demonstration of the tree-based prefetcher on 512 KB memory chunk for two different page access patterns. . . . . 17

5 Delayed page migration upon exceeding a static access threshold or exclusive data migration on write. . . . . 19

6 Transfer rate improves with transfer size as observed on real hardware with PCI-e 3.0 16x interconnect. . . . . 32

7 Comparing kernel execution time with different software prefetching schemes against no prefetching. . . . . 35

8 Performance validation of UVMSmart against real hardware. . . . . 42

9 Demonstration of tree-based pre-eviction on 512 KB memory chunk. . . . . 45

10 Comparing the effect of different eviction policies on kernel execution time. Tree-based prefetcher is active before reaching device memory capacity. Upon oversubscription, software prefetcher is disabled and $4KB$ pages are migrated on-demand. Working set is 110% of the device memory size. . . . . 47

11 Comparing total number of pages evicted for different eviction schemes. . . . . 48

12 Comparing the effect of different combinations of eviction policies and software prefetcher after oversubscription on kernel execution time. Tree-based prefetcher is active before reaching device memory capacity. Working set is 110% of the device memory size. . . . . 49

13 Page access pattern of nw benchmark without eviction. . . . . 50

14 Sensitivity of combinations of tree-based prefetcher and pre-eviction to the percentage of memory over-subscription by the working sets. . . . . 51
15 Effect of reserving a certain percentage of pages of LRU list from eviction on kernel runtime. Working set is 110% of the device memory size. Tree-based prefetcher is active before reaching device memory capacity. . . . . 52
16 Comparing the performance of tree-based pre-eviction against 2MB large page eviction. . . . . 53
17 Comparing the effect of tree-based pre-eviction and 2MB large page eviction on the total number of pages thrashed. . . . . 54
18 Visualizing page access distribution detailing type of access and total number of accesses per page per managed allocation for fdtd and sssp. . . . . 56
19 Visualizing page access patterns of a regular (fdtd) and an irregular (sssp) application over two iterations. (a), and (b) show access pattern of fdtd in iterations 2, and 4 respectively. (c), and (d) show access pattern of sssp in iterations 3, and 5 respectively. . . . . 57
20 Sensitivity of workloads to the static access counter threshold for delayed migration. . . . . 63
21 Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration scheme under no memory oversubscription. . . . . 64
22 Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes. . . . . 65
23 Comparing the impact of dynamic access counter based adaptive scheme on memory thrashing against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes. . . . . 66
24 Sensitivity of workloads to the multiplicative migration penalty. . . . . 67
25 Performance variation between LRU and LFU page replacement strategies. . . . 68

26 Comparing performance of schemes where (i) $2MB$ blocks are always written back and (ii) only dirty pages are written back and clean pages are invalidated directly. . . . . 69
27 An example of the hierarchical data-structure keeping track of block migration addresses used by detection engine. . . . . 74
28 Deterministic Finite Automaton (DFA) for managed allocations demonstrating the transition of migration states. . . . . 75
29 Performance of smart adaptive framework compared to unified runtime . . . . . 77
30 Categorizing workloads based on memory access pattern, arithmetic intensity, and number of DMA transactions per unit memory. . . . . 80
31 Effect of application consolidation on system throughput. FR-FCFS is the default scheduling policy. . . . . 82
32 Different performance slowdowns experienced when different interconnect scheduling schemes are employed. . . . . 83
33 Unwanted page eviction and in turn performance slowdown by application consolidation under device-memory oversubscription. App-1 and App-2 are respectively high- and low-priority application chosen by adaptive host-to-device interconnect scheduler. . . . . 85
34 The effect of adaptive CPU-GPU interconnect scheduling on weighted speedup for 17 representative workloads. . . . . 92
35 The summary of instruction throughput from adaptive CPU-GPU interconnect scheduling for all 45 workloads. . . . . 93
36 The summary of harmonic speedup from adaptive CPU-GPU interconnect scheduling for all 45 workloads. . . . . 94
37 The effect of priority preserving device-to-host CPU-GPU interconnect provisioning on weighted speedup and page eviction for 6 representative workloads. . 96

# 1.0 Introduction

Today, heterogeneity in memory technology and core types is ubiquitous in systems starting from hand-held smartphones to large supercomputers and commodity cloud platforms. For example, Oak Ridge National Laboratory's Titan supercomputer [77] incorporates graphics processing units (GPUs) and Intel's Xeon Phi co-processors alongside traditional central processing units (CPUs). Similarly, Amazon Web Services (AWS) [6], one of the largest cloud providers, offers instances of CPU-GPU platforms built with Intel CPUs and NVIDIA GPUs. The union of high thermal design power (TDP) processors in heterogeneous systems offer new performance opportunities for applications. For example, while serial code sections can run efficiently on ILP-optimized CPU processors, parallel code with fine-grained data parallelism benefit from running on accelerators such as GPUs resulting in aggregate savings of millions of dollars in large-scale systems.

While some of these heterogeneous systems may share a single homogeneous pool of physical memory between CPUs and GPUs, discrete GPUs connected with x86 processors via peripheral component interconnect express (PCIe) dominate the marketplace. As the number of scalar cores and SIMT (Single Instruction Multiple Threads) units in GPUs continues to grow, memory bandwidth is also scaled proportionately to keep the compute-resources busy. However, GPU memory capacity remains relatively smaller compared to the capacity of CPU-attached memory. For example, while CPUs are likely to continue using cost and capacity-optimized DRAM (DDR4, LPDDR4) technology, GPUs are moving towards using capacity-limited, but bandwidth-optimized, on-chip memory packages such GDDR5, High Bandwidth Memory (HBM), and Wide-IO2 (WIO2). Due to the large differences in bandwidth and capacity of the individual memory modules, memory management becomes challenging with respect to the system design and programmability of discrete CPU-GPU systems.

## 1.1 Problem Description

Due to discrete physical memory modules, traditionally application programmers had to explicitly call memory copy APIs to copy pre-initialized data over the relatively slow CPU-GPU interconnect (e.g. PCIe) to the GPU's physical memory before launching GPU kernels. This upfront memory transfer is an important aspect while quantifying GPU performance because, for long-running GPU kernels, this bandwidth-optimized bulk-transfer amortizes the migration overhead. However, relatively smaller GPU memory capacity restricts the effective working sets of the GPU programs. As a result, the onus of memory management falls squarely on the programmers. Application developers are forced to tile their data for migration and painstakingly launch GPU kernels over multiple iterations. This burden has been considerably relaxed by the introduction of Unified Virtual Memory (UVM). To this date, stand-alone PCIe-attached GPUs are treated as slave accelerators. The runtime, loaded as a set of kernel modules in the host operating system, is the key to tap into the computation capabilities of a GPU. NVIDIA introduced software-managed runtime which provides the illusion of unified memory space by providing a virtual memory pointer shared between CPU and GPU. With the assistance from hardware page-faulting and migration engine, UVM automates the migration of data in and out of the GPU memory even upon device-memory over-subscription.

Although fault-driven migration and non-blocking, outstanding, replayable page faults in unified memory improve programmability, it is not sufficient. As GPU's massive thread-level parallelism (TLP) can no longer mask the latency of migrating pages over slow CPU-GPU interconnect, researchers have felt the need for prefetching pages in advance to overlap computation with the migration of future-referenced pages. The concept of prefetching is not new in hierarchical memory systems. Prefetchers are designed to exploit spatial- and/or temporal-locality of memory accesses of prevalent workloads to reduce the amount of time a computation pipeline is stalled for the availability of data/operands. In the past, researchers had explored both micro-architectural and software-based prefetchers [8, 29, 30, 34, 52, 54, 72]. However, prefetching pages between host and device memory over CPU-GPU interconnect stands out due to several unique systems properties and performance requirements in con-

trast to the hierarchical memory models in traditional multi-core symmetric multi-processor (SMP) systems. Firstly, when the working set of a GPU workload fits in the GPU device memory, aggressive prefetching has little to no downside as performance always benefits from bandwidth-optimized access to local memory. However, for applications with sparse, random, and seldom access over large memory allocations, prefetching pages indiscriminately can quickly lead to device-memory oversubscription. Moreover, because of the large number of SIMT units and massive TLP, GPU compute units (CUs) are constantly generating on-demand requests for page migrations. As a result, a prefetcher cannot afford to flood the DMA engine with prefetch requests and in turn throttle on-demand transfers. Secondly, GPU has been traditionally used as a slave to the host processor. The software-managed runtime in the host operating system is responsible for memory management as well to manipulate GPU's page table. As a result, hardware-assisted prefetching is not a viable solution for prefetching data from the host to the device memory. Similarly, considering the requirement of higher programmability and application-transparency, user-assisted and/or compiler-directed prefetchers are not preferred for GPU workloads.

Acknowledging these unique challenges for prefetching presented in the CPU-GPU memory hierarchy, researchers have proposed the concept of software prefetchers [94, 2]. Ganguly at al [32], through their extensive micro-benchmarking and investigation of the NVIDIA UVM module, showed that there exists a tree-based software prefetcher implemented in the CUDA unified memory runtime. Implementing a prefetcher in GPU runtime exposes new opportunities by taking away the limitations of both hardware and user-directed prefetching. Because GPU interrupts the runtime hosted on CPU and communicates the pending faults, the runtime is in a unique position of maintaining a historic view of the memory accesses and making an informed choice of selecting a prefetch candidate based on the spatio-temporal locality of access.

However, when GPU memory has no free space to allocate pages for newer migration, aggressive prefetching can be counterproductive. Runtime, upon detecting memory-oversubscription, needs to evict pages to allow not only on-demand migration but also insertion of prefetch candidates. NVIDIA's unified memory runtime implements an LRU page replacement with $2MB$ huge-page eviction granularity. Like aggressive prefetching, aggressive

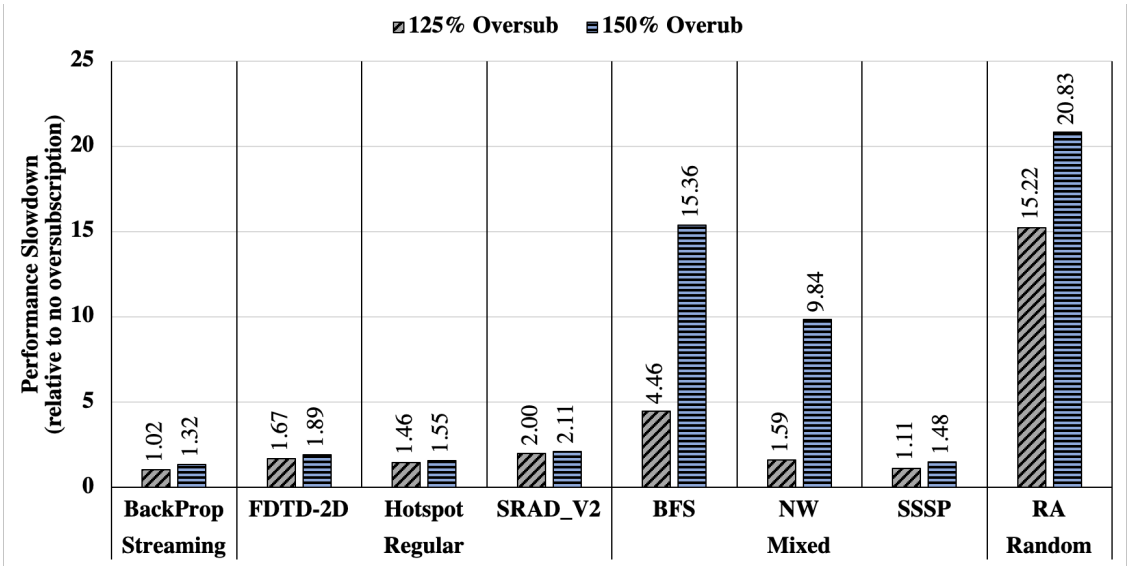



Figure 1: Sensitivity of workloads to the percentage of memory oversubscription (performed on real hardware).

eviction has an adverse effect on performance. Figure 1 shows the performance degradation of GPGPU workloads with varying percentages of memory oversubscription. These workloads are described in Section 3.2. The results are obtained by running the workloads on `GeForceGTX 1080 ti` [58] *(not on simulated environment)*. Eviction with huge-page granularity causes a large page thrashing for repetitive kernel launches. This necessitates careful investigation and design of a new locality-aware page replacement strategy compatible with locality-aware software prefetcher.

In recent years, discrete CPU-GPU heterogeneous systems have been moving away from the PCIe interface. By layering coherence protocols on top of physical link technologies (e.g. NVLink, Hypertransport, etc.), these systems achieve high bandwidth and low latency between the NUMA pools attached to GPU and CPU respectively. As a result, CPU-GPU heterogeneous systems are closely resembling the traditional CC-NUMA and even hierarchical hybrid memory systems. Today, GDDR5 is the most common bandwidth-optimized memory technology used with discrete GPUs. Due to the high data rates (per-pin data

rate up to 7Gbps), GDDR5 requires significant energy per access and in turn, cannot scale into high-capacity multi-rank memory modules. Whereas, CPU-attached DRAM modules (DDR4, LPDDR4) can provide similar latency at a much lower energy requirement per access but fail to provide a high data rate (only 3.2 Gbps per pin).

Traditionally, operating systems characterize NUMA zones based on the difference in respective memory access latency and further assume that these NUMA zones will be symmetric in bandwidth and power characteristics. However, in the context of CPU-GPU heterogeneous systems, this assumption clearly breaks down. The situation is further exacerbated as GPU-runtime is unaware of the potential impact of these differences in memory characteristics on the performance and energy requirements of the running applications. The disaggregation of memory into on- and off-package pools and significant variation in energy requirement, bandwidth, and latency between the discrete memory pools are two primary factors motivating the necessity to revisit page placement and pinning decision in the context of CPU-GPU Unified Memory. Moreover, with heterogeneous computing, certain phases of computation are pinned to either CPU or GPU. Unlike traditional NUMA-aware SMP systems, processes are not migrated to the data they are operating on to mitigate performance imbalance. As a result, it is runtime's responsibility to decide where the memory pages to be pinned and when to be migrated in case of heterogeneous CPU-GPU systems. The goal of a robust page placement and pinning strategy is to abstract the technical properties of on- and off-package memory into power and performance characteristics based on which an optimization decision can be made.

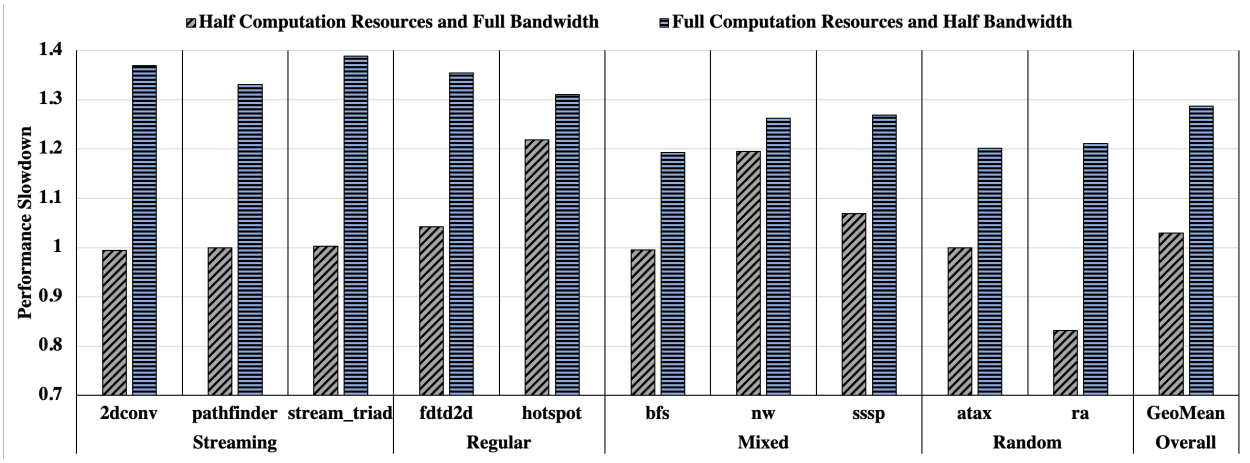
Beyond the peculiarities of memory characteristics and asymmetry in bandwidth, access latency, capacity, and power consumption, memory access pattern of GPU workloads plays vital roles while deciding page placement and pinning. Agarwal et al [3] showed that over 60% of the memory bandwidth stems from within only 10% of the application's allocated pages. This is due to the fact that distinct ranges of physical addresses appear to be clustered as hot or cold (based on the aggregate access frequency) for a wide range of irregular data-intensive workloads. Further, migrating a cold page can cause serious thrashing of the heavily accessed or hot page which is the primary cause of oversubscription overhead for irregular data-intensive applications with sparse, random, and seldom access over large cold

data sets. This necessitates the exploration of a user-agnostic page placement and pinning policy in the software-managed runtime leveraging existing hardware and software support for data-intensive workloads in CPU-GPU heterogeneous systems.

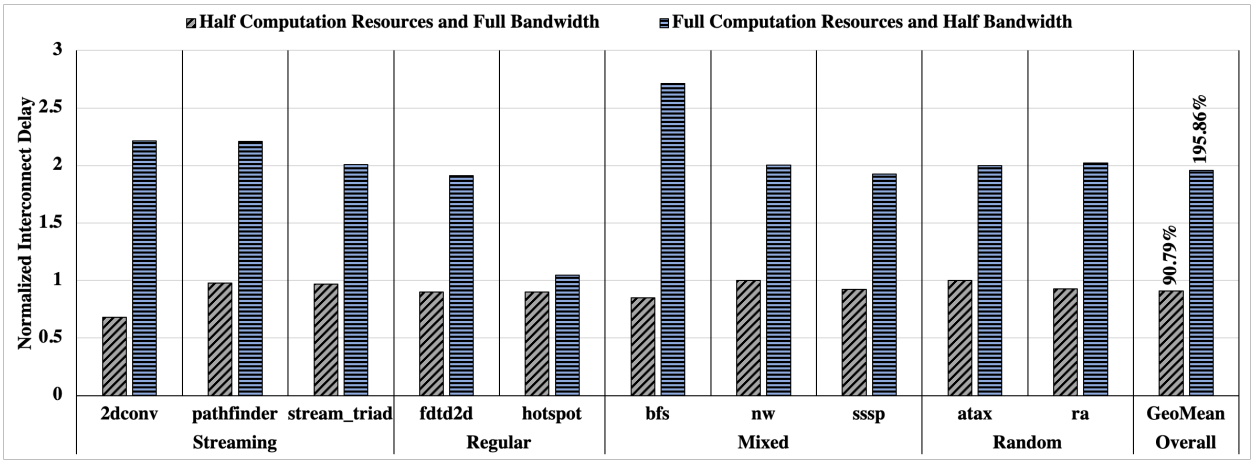
The memory access pattern of a GPU application depends on the fine-grained parallelism and inherent memory access characteristics. Different access patterns can react differently to memory over-subscription. Detailed analysis by runtime profiling shows that fault-based migration under oversubscription waits for long latency writebacks in case of *regular* applications. On the other hand, the oversubscription overhead in *irregular* applications is due to excessive page thrashing which is further exacerbated by the prefetcher. *Irregular* applications show an order of magnitude performance degradation. Although CUDA provides a wide array of APIs to properly allocate and manage a working set of general-purpose applications, taking full advantage of smart memory management tactics depend on the ninja programming skills of application developers. This necessitates the design of a smart, adaptive runtime that can monitor and determine the underlying pattern in CPU-GPU interconnect traffic and then choose and apply the best possible memory management strategy transparent to the application developers.

In today's heterogeneous installations, a single GPU is often shared by multiple applications potentially originating from different users to fully saturate available compute-resources of the GPU. Researchers [59, 60, 76, 82, 89, 1] have shown that executing concurrent applications on the same GPU improves overall system throughput. However, workload consolidation suffers from the problem of application interference contending for various system resources - memory hierarchy, on-chip interconnect bandwidth and compute cores. Past researches [1, 4, 5, 36, 86, 59] have investigated the issue with contention for compute resources by concurrent GPU applications. Only a few works [42, 43] have explored the application interference in the on-chip interconnect and device memory.

Figure 2a reports the performance slowdown of GPU applications when provisioned - (i) 50% of available SMs with full share of interconnect bandwidth and (ii) 50% of interconnect bandwidth but executed on all available SMs compared to the case where the respective application runs with full compute resources and bandwidth. We can see that halving the number of SMs only causes 3% (geometric mean) slowdown compared to 29% slowdown




(a) On end-to-end execution time




(b) On average delay suffered by CPU-GPU interconnect traffic

Figure 2: The effect of computation resources and interconnect bandwidth on unified memory applications.

with half of available interconnect bandwidth. Note that most of the applications show little to no slowdown from 50% reduction in the number of SMs. This behaviour might appear counter-intuitive. However, most of these algorithms use more threads-per-blocks and fewer blocks-per-grid to fully saturate the scheduling capacity per SM. Hence, they cannot fully scale to multiple SMs leading to poor utilization of GPU compute resources. Moreover, `ra` gains performance. `ra` creates bursts of random requests per thread. Halving the number of

SMs leads to fewer active threads and in turn lesser contention for interconnect bandwidth. This is similar to the method of thread-throttling to remove memory interference in chip-multiprocessors [28, 23]. This validates the use case of sharing GPU compute resources in a spatial manner.

Figure 2b demonstrates that the normalized delay suffered by CPU-GPU interconnect traffic is almost twice when the interconnect bandwidth is halved and has no significant impact when half of the total SMs is provisioned. This justifies the performance behavior reported in Figure 2a. 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 to hide long latency host-to-device page migrations. As a result, CPU-GPU interconnect becomes a performance-critical resource when multiple concurrent applications, sharing the same GPU, contend for it. This motivates the particular use case of user-agnostic CPU-GPU interconnect provisioning for concurrent applications in commodity clusters.

## 1.2 Thesis Statement

*"Unified Memory provides greater programmability by simplifying memory management in CPU-GPU heterogeneous systems. Despite significant advancement in memory management techniques, over-subscription remains a major challenge in the adaptation of GPUs by data-intensive applications. Moreover, the applicability of certain memory management techniques still relies on explicit programming hints supplied by application developers. Further, multi-tenancy in the GPU cloud is not explored to its full potential. The goal of this dissertation is to design an adaptive memory management system, which can cater to the performance goals of a large, and disparate set of general-purpose applications using Unified Memory in CPU-GPU heterogeneous systems."*

## 1.3 Contributions

The primary goal of this dissertation is to design a smart, application-transparent runtime that can improve GPU programmability and also address the performance requirements of disparate general-purpose applications. To this end, this dissertation attempts to answer the following research questions.

- **RQ-1.** How to mitigate performance overhead from device-memory oversubscription by the application working set?
  - **RQ-1.1.** Is it possible to design a page replacement strategy that adopts the semantics of existing software prefetcher?
  - **RQ-1.2.** How to leverage existing hardware and software support to characterize memory pages based on access-frequency, and make smart pinning and migration decision?
  - **RQ-1.1.** How to transparently identify any underlying pattern in CPU-GPU interconnect traffic and apply suitable memory management policies to deal with oversubscription overhead?
- **RQ-2.** How to ensure overall system throughput while guaranteeing performance fairness by eliminating interference in multi-tenant execution?

To this end, the main contributions of this dissertation can be outlined as follows.

1. Adaptive Page Replacement
   - Adopts the semantics and leverages the implementation of the existing tree-based software prefetcher.
   - By following the spatio-temporal locality, trades between 4KB and 2MB page eviction granularity.
2. Adaptive Page Migration and Pinning
   - Pins hot pages to the device memory and cold pages to the host memory.
   - Balances between high-bandwidth, sequential access to device-local memory and low-latency, sparse, remote access to the host memory.

3. A Smart Unified Memory Management Framework
    - Detects underlying pattern in CPU-GPU interconnect traffic.
    - Unifies memory management by dynamically and transparently employing the best-suited memory management policy to reduce page thrashing.
4. Adaptive Multi-tenancy
    - Extends the characterization of unified memory applications beyond device memory access pattern by introducing arithmetic intensity and the number of interconnect traffics per unit memory as two new classification metrics.
    - Develops a strong analytical model that expresses application performance as a function of their arithmetic intensity and number of on-demand memory transactions.
    - Based on the theoretical model, presents an application-aware, adaptive interconnect provisioning scheme that aims to - (i) improve overall system throughput, and (ii) guarantee application fairness by fair-share, work-conserving scheduling.

In addition to the main contributions listed above, the following contributions were necessary to enable the design and evaluation of the presented work.

1. Discovery of the Semantics of Tree-based Prefetcher
    - Creating micro-benchmarks to reverse engineer the semantics of tree-based prefetcher
2. Evaluation Framework for Isolated Execution
    - Extending of GPGPU-Sim 3.x to provide functional and timing simulation support for Unified Memory
    - Introducing new benchmarks with Unified Memory APIs
3. Evaluation Framework for Concurrent Execution
    - A Unified Memory Concurrent Application (UMCA) suite and a simulation framework to support execution of such application suite

## 1.4 Organization

Figure 3 gives an overview of the dissertation. Chapter 2 provides the necessary background to navigate the rest of the dissertation along with relevant prior work motivating the proposed contributions. The extended simulation framework and new unified memory benchmarks are explained in Chapter 3. Chapter 4 presents the contribution on adaptive page replacement (published in ISCA 2019 [32]). Chapter 4 describes the heuristic for adaptive page migration and pinning (published in IPDPS 2020 [31]). Chapter 6 introduces a smart, adaptive extension to the unified memory runtime that transparently detects the page-migration pattern and applies memory management techniques to holistically address device-memory oversubscription. Chapter 7 presents an adaptive CPU-GPU interconnect scheduler for multi-tenant GPGPU workloads in CPU-GPU heterogeneous memory systems. Finally, Chapter 8 summarizes this dissertation by highlighting the expected impact along with the future research direction.

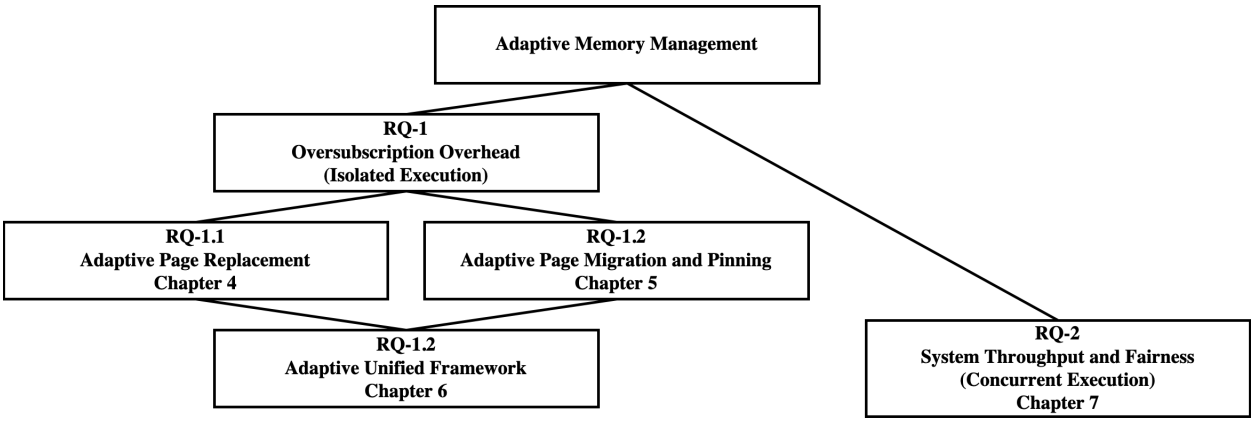



Figure 3: Overview of the thesis work.

# 2.0 Background and Related Work

This chapter provides relevant background information on memory management in CPU-GPU heterogeneous systems setting the necessary foundation for the following chapters. The description closely follows NVIDIA/CUDA terminology in specific cases, however, it is general enough to describe any vendor-agnostic discrete CPU-GPU heterogeneous memory system. It also includes a summary of relevant prior research that motivates the contributions in the following chapters.

## 2.1 Baseline Architecture

This dissertation considers discrete GPUs connected to the host CPU over PCIe (peripheral component interconnect express) as they dominate the marketplace over the integrated on-die GPUs. Multiple compute cores are grouped in Streaming Multiprocessors (SMs) along with registers, shared memory, and caches. The GPU has many SMs. SMs are connected to multiple Memory Controllers (MCs) via an on-chip interconnect network and share the device memory (e.g. GDDR5, HBM, etc.).

GPUs are co-processors and governed by software runtime. The software runtime executes as a device driver part of the host operating system (OS). The host application launches CUDA computation kernels. Each kernel is organized as blocks (TBs) of co-operative thread arrays (CTAs). Once a TB is dispatched to an SM, its threads are batched into warps that are scheduled on the SM.

## 2.2 Unified Memory

Traditionally, the onus of memory management and utilizing overall system bandwidth had fallen squarely on application programmers. In the classic, "copy then execute" model,

application programmers had to allocate memory on both host and device memory pools, then copy the data from the host to device memory before launching the GPU kernels, and then finally copy the data back from the device to the host memory pool. Programmers also had to painstakingly tile the working set to work around memory oversubscription and write smart multi-stream asynchronous constructs for data migration to overlap with kernel execution.

To address these challenges with memory management and accommodate increasing memory demand of general-purpose applications, GPU vendors introduced device memory virtualization. Although the Heterogeneous System Architecture (HSA) Foundation has its version of unified virtual addressing (UVA), NVIDIA GPUs are ahead in their game due to their hardware support for fault-driven migration and on-demand allocation powered by software-runtime for unified memory. By automatically paging in and out of device memory, unified memory simplifies memory management, and improves programmability.

### 2.2.1 Fault-driven Migration and On-demand Allocation

The key to realizing the illusion of a unified address space is micro-architectural support for fault-driven migration and on-demand allocation. NVIDIA Pascal GPUs [56] have introduced hardware page faulting and Page Migration Engine to support Unified Memory for discrete CPU-GPU systems. In CUDA 8.0 [55], `cudaMallocManaged` allows programs to allocate data that can be accessed by both host code and kernel using a single shared pointer. The illusion of Unified Memory is realized by on-demand allocation and fault-driven data transfer.

In the "copy then execute" model, the host program ensures that data is physically available in the device memory before the kernel starts executing. Warps are stalled on *near-faults* which occurs only upon L2 cache misses. The massive thread-level parallelism (TLP) hides the local memory access latency and guarantees high throughput. However, in Unified Memory, a new type of fault, which will be referred to as *far-faults*. Zheng et al [94] introduced the concept of replayable far-fault. A far-fault occurs when the addressed memory page is not physically present in the device memory. On-demand allocation and

page migration is triggered by these far-faults.

The following steps demonstrate the lifetime of a far-fault resolution. ① Scheduled warps generate global memory accesses. ② Each Streaming Multiprocessor (SM) has its own load/store unit. Every load/store unit has its L1 TLB. The load/store unit performs a TLB lookup to find whether the translation for the issued memory access is cached or not. ③ There is also an L2 TLB shared across all SMs of a GPU. Missed translation requests in L1 TLB are searched through L2 TLB. ④ A L2 TLB miss is relayed to the GPU Memory Management Unit (GMMU). ⑤ The GMMU walks through the page table looking for a page table entry (PTE) corresponding to the requested page with the valid flag set. A far-fault occurs if there is no PTE for the requested page or the valid flag is not set. ⑥ GMMU can only perform a page table lookup. It has no capability to handle far-fault 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. ⑦ GPU interrupts runtime hosted by the host operating system to relay the far-faults every set quantum. ⑧ Unified memory runtime groups the far-faults. The software prefetcher (detailed in Section 2.2.2) determines and schedules direct memory access (DMA) request to transfer a chunk of memory from the host to the device over CPU-GPU interconnect. ⑨ Pages corresponding to the memory chunk are allocated on-demand, and page table and TLB entries are created/updated upon completion of the scheduled DMA request. ⑩ The MSHRs are consulted to notify the corresponding load/store unit and the memory access is replayed. The offending warps are marked executable.

### 2.2.2 Tree-based Software ($TBN_p$) Prefetcher

GPU Technology Conference 2018 [70] briefly mentioned a tree-based hardware prefetcher implemented by NVIDIA CUDA 8.0 driver. Knowledge of the exact semantics of this prefetcher is proprietary to NVIDIA and was never made public. One of the key contributions of this dissertation is the discovery of the semantics of the tree-based software prefetcher by studying the MIT-licensed open-source `nvidia-uvm` module, and executing micro-benchmarks described in Section 3.2.1 on `GeForceGTX 1080ti` and profiling the mem-

ory accesses using `nvprof`.

The semantics of $\text{TBN}_\text{p}$ demands that every `cudaMallocManaged`ed allocation is first logically divided into $2MB$ huge-pages. Then, these $2MB$ huge-pages are further divided into logical $64KB$ basic blocks to create a full binary tree (or a proper binary tree or a 2-tree) per huge-page boundary. By the definition of a full binary tree, every node has exactly two children nodes. The root node of each binary tree corresponds to the virtual address of a $2MB$ large page and the leaf-level nodes correspond to the virtual addresses of the $64KB$ basic blocks. If the user-specified size of the allocation is not a perfect multiple of $2MB$, then the remainder size of the allocation breaks the principle of a full binary tree. To address this, the remainder allocation is rounded up to the next $2^i * 64KB$ and another full binary tree is created. For example, if the programmer specifies $4MB$ and $168KB$ size for a `cudaMallocManaged` allocation, at the time of allocation, the runtime rounds this size up to $4MB$ and $256KB$. Then two full binary trees for $2MB$ large pages and one full tree for $256KB$ are created and maintained by the runtime transparent to the programmer's knowledge. This behavior can also be verified by running the micro-benchmarks described later.

The maximum memory capacity of a node in the full binary tree can be calculated as $2^h * 64KB$, where $h$ is the height of a node and $h = 0$ at the leaf level. On every far-fault, the runtime first identifies the $64KB$ basic block corresponding to the faulty page being requested. With the understanding that upon migrating, 16 pages in the basic block will be validated in the GPU page table, runtime then recalculates the to-be valid size of its parent and grand-parent up to the root node of the tree. Here and henceforth, the valid size means the size of all valid pages corresponding to the leaf-nodes belonging to a given node. At any point, if runtime discovers the to-be valid size of a node is strictly greater than 50% of the maximum memory capacity at this level, it tries to balance the valid sizes between the two children of that node. This balancing process is recursively pushed down to the children which have not reached the maximum valid size quota. This balancing act identifies basic blocks for prefetching. This process continues till no more basic blocks at leaf level can be identified as prefetch candidates and the to-be valid size of any non-leaf node including root is not more than 50% of maximum size capacity at its level.

Prefetching contiguous pages within $2MB$ boundary tries to ensure the allocation of larger contiguous memory and can also help bypass traversing the nested page tables. This helps reduce the time to access memory. For this same reason, in their work [10], researchers introduced the concept of memory defragmentation to swap and coalesce fragmented memory chunks to ensure contiguous physical memory worth of $2MB$ large page. However, migrating $4KB$ pages on-demand and then defragmenting the memory space in the runtime has substantial overhead. Whereas, $\text{TBN}_\text{p}$ is an adaptive scheme where the prefetch size can vary from $64KB$ to $1MB$ based on the access pattern and opportunity of prefetching. Thus, it can get close to $2MB$ large page locality without causing any additional performance overhead.

$\text{TBN}_\text{p}$ can be demonstrated with the help of two examples in Figure 4. Both of these examples explain the semantics on $512KB$ memory chunk for simplicity. These examples use $N_h^i$ to denote a node in the full binary tree, where $h$ is the height of the node and $i$ is the numeric position of the node in that particular level. It is further assumed that initially all pages in this $512KB$ allocation are invalid with valid bit not set in the GPU's page table and thus every first access to a page causes a far-fault.

In the first example, for the first four far-faults, runtime identifies the corresponding basic blocks $N_0^1$, $N_0^3$, $N_0^5$, and $N_0^7$ for migration. In this example, as the first byte of every basic block is accessed, the basic blocks are split into $4KB$ page-fault groups and $60KB$ prefetch groups. All memory transfers are serialized in time. After these first four accesses, each of nodes $N_0^1$, $N_0^3$, $N_0^5$, and $N_0^7$ has $64KB$ valid pages. Then, runtime traverses the full tree to update the valid page size for all the parent nodes and thus each node at $h = 1$ ($N_1^0$, $N_1^1$, $N_1^2$, and $N_1^3$) has $64KB$ valid pages. When the fifth access occurs, runtime discovers that $N_1^0$ and $N_2^0$ will have $128KB$ and $192KB$ valid pages respectively. For $N_2^0$, the to-be valid size is greater than 50% of the maximum valid size of $256KB$. Hence, the right child $N_1^1$ is identified for prefetching. This decision is then pushed down to the children. This process identifies the basic block $N_0^2$ as a prefetch candidate. Further, runtime discovers that after prefetching $N_0^2$, $N_3^0$ will have $320KB$ of valid pages which is more than 50% of the maximum valid size of $512KB$. Then, node $N_3^0$ pushes prefetch request to the node $N_2^1$ which in turn pushes it to its children. This process identifies basic blocks $N_0^4$ and $N_0^6$ for

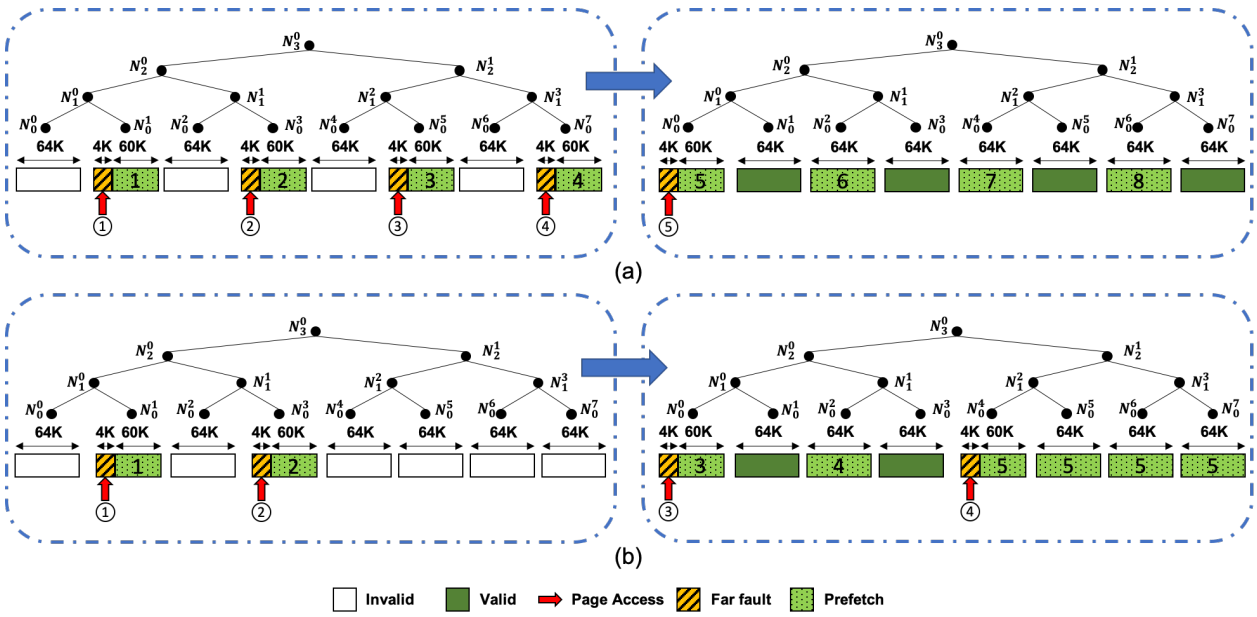



Figure 4: Demonstration of the tree-based prefetcher on 512 KB memory chunk for two different page access patterns.

further prefetching.

In the second example, the first two far-faults cause migration of basic blocks $N_0^1$ and $N_0^3$. runtime traverses the tree to update the valid size of nodes $N_1^0$ and $N_1^1$ as $64KB$ each. At the third far-fault, as basic block $N_0^0$ is migrated, the estimated valid sizes for nodes $N_1^0$, and $N_2^0$ are updated as $128KB$ and $192KB$ respectively. As the valid size of $N_2^0$ is more than 50% of the maximum valid size of $256KB$, $N_0^2$ is identified for prefetching. After this point, the $N_2^0$ is fully balanced and both $N_2^0$ and $N_3^0$ have exactly $256KB$ of valid pages. On fourth access, runtime discovers that the valid size of $N_3^0$ will be $320KB$ which is more than 50% of the maximum memory size it can hold. This imbalance causes prefetching of nodes $N_0^5$, $N_0^6$, and $N_0^7$. Note at this point as runtime finds four consecutive basic blocks, it groups them to take advantage of higher bandwidth. Then, based on the page fault, it splits this $256KB$ into two transfers: $4KB$ and $252KB$. An interesting point to observe here is that for a full binary tree of $2MB$ size, $\text{TBN}_\text{p}$ can prefetch at most $1020KB$ at once in a scenario

similar to the second example.

### 2.2.3 Page Replacement

One of the major benefits of unified memory is that the runtime automatically evicts older pages to make room for the newer page migrations taking care of the device-memory oversubscription. Before scheduling any DMA request to copy memory pages 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. As the pages are migrated in, they are placed in a queue based on the migration timestamp. After migration, if a page is accessed, then its position is updated based on the current access timestamp. Newly accessed pages are moved to the end of the queue and thus the oldest accessed (/migrated) page will be evicted upon oversubscription. The page replacement works at the strict granularity of $2MB$ x86-OS huge-page. A $2MB$ huge-page is selected for eviction only when it is fully populated and not currently addressed by any scheduled warp. Evicting $2MB$ ensures that the semantics of the tree-based prefetcher is not violated and thus stays active even after device memory oversubscription.

### 2.2.4 Remote Zero-copy Access and Delayed Migration

Unified Memory offers a "single-pointer-to-data" model. Both host and device see a unified view of virtual address space. At any given time, only one physical copy of the data is maintained either on the host or the device memory. Typically, the data is initialized in the host memory. On every first access to a page by the device, the corresponding page table entry in the host is invalidated and data is migrated to the device memory and a new entry is created in the device page table. On the contrary, with zero-copy allocations, the physical allocation is hard-pinned to the host memory. This means pages are never copied from the host to the device memory. Rather the device accesses data remotely over cache-coherent interconnect. `cudaHostRegister` API allows `malloc`ed allocation to

be pinned to the host memory and the kernels are launched with device pointer derived using `cudaHostGetDevicePointer` API. Remote zero-copy access has lower latency than the classic Direct Memory Access (DMA) but also suffers from lower bandwidth of PCIe interconnect. This is why zero-copy access is introduced for applications with seldom and sparse access to very large data sets. OpenCL [9, 7] also provides support for allocating host pinned memory using `CL_MEM_ALLOC_HOST_PTR`.

Following the same concept, CUDA 9.0 offers the ability to provide useful hints to the Unified Memory subsystem about the usage pattern. The `cudaMemAdviseSetAccessedBy` flag allows the device to establish direct mapping to the host memory. Further, the preferred location of memory allocation can be set using `cudaMemAdviseSetPreferredLocation`. However, the pages in the host memory are soft-pinned because based on runtime heuristics pages can be migrated to the local from the far memory.

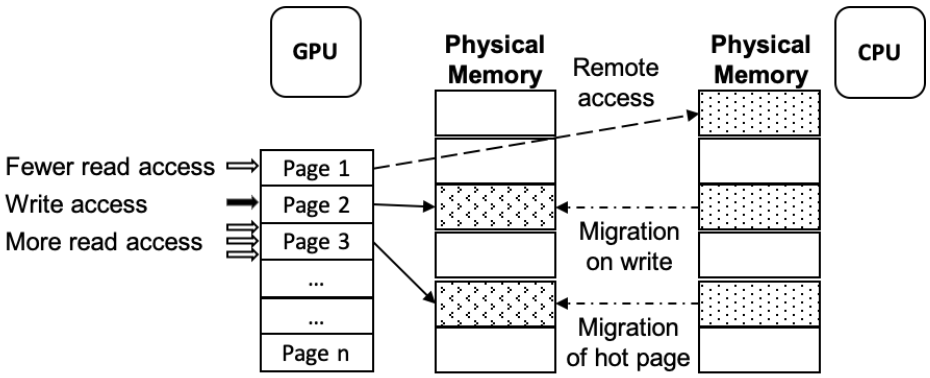



Figure 5: Delayed page migration upon exceeding a static access threshold or exclusive data migration on write.

NVIDIA Volta GPUs [71] and IBM Power9 [38] introduced a new hardware-based page-level access counter. If an allocation is advised to be soft-pinned to the host memory, then the memory is not copied directly at the first-touch by the device. Rather, the migration from the preferred location of host memory to the device memory is delayed based on a static access counter threshold, $t_s$. If the page is accessed to read data for a certain number of times crossing the value of $t_s$ configured in the driver, the data is copied to the device

memory. On the other hand, on write access, the page is invalidated in the host page table and exclusively copied to the device memory irrespective of the access frequency [70]. This access counter-based delayed page migration mechanism is illustrated in Figure 5.

Irregular applications with sparse memory access can highly benefit from both remote zero-copy access and access counter-based delayed migration. As in Unified Memory, fault-based migration triggers additional prefetching of neighbor pages, under a strict memory budget it can exacerbate the situation causing a crippling impact on performance. Delayed migration or no-copy can improve performance for irregular applications by reducing the number of page thrashing. However, for regular applications with dense, sequential access zero-copy is a bad option. Although the remote zero-copy model offers low latency of access, migrating data in bulk to the local memory and then accessing it enjoys the benefits of bandwidth optimized local network. Moreover, larger migration using prefetcher improves PCI-e bandwidth utilization and reduces the number of far-faults in general. Similarly, having static access counter-based threshold for delayed migration incurs the additional overhead of remote access because, for dense sequential access, the data is eventually migrated to the local memory upon crossing the threshold.

## 2.3 GPU Multi-tenancy

### 2.3.1 GPU Sharing Mechanism

The state-of-the-art GPU architecture provides software-based GPU multi-tenancy leveraging the hardware support of Hyper-Q [62]. A CUDA stream is a software abstraction of a sequence of commands that execute in order. All work on the GPU is launched either explicitly into a CUDA stream, or implicitly using a default stream. The default stream is a special stream that implicitly synchronizes with all other streams on the device. Whereas, different streams, other than the default stream, may execute their commands concurrently, allowing for coarse-grained parallelism. Starting from Kepler architecture, NVIDIA provides 32 work queues between the host and the GPU and a concurrent scheduler to schedule work

from work queues. CUDA streams are aliased onto one or more *work queues* on the GPU by the driver. Thus, streams enable concurrent kernel execution on GPU which in turn facilitates better utilization of GPUs and overall performance.

NVIDIA released an alternative, binary-compatible implementation of the CUDA API called Multi-Process Service (MPS) [57]. The MPS client runtime is built into the CUDA driver library which can be used transparently by multiple CUDA applications to utilize Hyper-Q capabilities of the GPU. The MPS server is the clients' shared connection to the GPU and provides concurrency between the clients. In contrast to the pre-Volta MPS, Volta MPS clients can submit work directly to the GPU bypassing the MPS server. While availing stream-based multi-tenancy needs code transformation, MPS allows dynamic scheduling of client codes.

### 2.3.2 CPU-GPU Interconnect Provisioning

Currently, the Peripheral Component Interconnect Express (PCIe) interface is commonly used to connect a GPU to a computer system. PCIe standard [39] includes the features of Virtual Channels (VCs) and Traffic Classes (TCs) for network traffic provisioning. Traffic classes indicate the priority of traffic, while Virtual Channels deliver the priority. Virtual channels allow higher-priority traffic to flow past lower priority traffic at every link. Transactions are associated with one of the supported VCs according to their TC attribute through TC-to-VC mapping, specified in the configuration block of the PCIe device. VCs have dedicated physical resources - buffering, flow control management, etc. As of today, the vast majority of GPU runtimes do not support interconnect provisioning for concurrent GPU applications. Currently, the best approach implemented in drivers is to serve the requests from concurrent applications based on a first-ready first-come-first-serve (FR-FCFS) basis. Starvation is avoided by load balancing between the application following round-robin (RR) ordering between transactions.

## 2.4 Related Work

### 2.4.1 Prefetching and Page Replacement in Unified Memory

The key to realizing the illusion of unified virtual address space is fault-driven migration and on-demand memory allocation. Zheng et al [94] 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 on cache-miss. Despite the higher programmability and ease of memory management, unified memory is still in its infancy. Agarwal et al [2] and Zheng et al [94] were one of the first few to demonstrate the performance bottleneck of on-demand paging and proposed software prefetchers as a prospective solution. Zheng et al [94] introduced (i) random, (ii) sequential, and (iii) locality-aware software prefetchers. Following these researches, NVIDIA's unified memory runtime introduces a tree-based spatio-temporal prefetcher as described in Section 2.2.2. Like prefetching, page replacement also plays a critical role in unified memory particularly under device memory oversubscription. Zheng et al [94] compared the performance difference from (1) random and (ii) LRU page eviction. Ganguly et al [32] studied the interplay between prefetchers and page replacement algorithms under oversubscription. They proposed a new "pre-eviction" policy inspired by the semantics of NVIDIA's tree-based prefetcher to reduce page thrashing which is one of the main contributions of this dissertation as presented in Chapter 4. Following the same direction, Yu et al introduced hierarchical page eviction [91], and a coordinated prefetcher and eviction policy [90].

### 2.4.2 Page Migration and Pinning

Over the years, the modern-day SMP systems have become more complex in their structure and component designs. Now, they are typically made of multiple cache-coherent non-uniform memory access (CC-NUMA) zones where each NUMA zone comprises of a socket, the processors within it, and the attached physical memory module. While a processor within a NUMA zone can freely access the memory from another NUMA zone to take advantage of the aggregate memory capacity and bandwidth, it comes at the cost of

non-uniform access latency. To mitigate the performance imbalance and benefit from data-locality, researchers have explored the idea of page placement and pinning in traditional CC-NUMA systems. Earlier works mostly focused on placing data and processes in close proximity [16, 17, 40, 49, 80, 87]. Acknowledging the fact that it is typically better for processes to service memory requests from their respective NUMA zone, operating systems like LINUX exposes the system topology and memory latency information using System Resource Affinity Table (SRAT) and System Locality Information Table (SLIT). Discovering this information, the application can allocate and place physical memory pages using `malloc` and `mmap` calls. There is another body of work [11, 15, 24, 26, 46, 75, 95], which considers sharing patterns, interconnect congestion, and even queuing delay within the memory controller as metrics to design page and process placement policies. Page placement and pinning also become crucial considerations for hybrid and hierarchical memory systems consisting of capacity-optimized non-volatile memory (NVM) alongside latency-optimized small DRAM modules. Several works [13, 20, 47, 53, 61, 63, 67] had studied the performance peculiarities of NVM in particular read/write latency disparity and focused to balance performance needs with power consumption. Although the above works explored heterogeneity in traditional SMP systems, and in hierarchical and hybrid-memory systems, the key idea of data-locality and memory placement motivated the work presented in Chapter 5.

While CPUs are generally more performance-sensitive to memory system latency rather than other memory characteristics, due to their massive TLP, GPUs can gracefully handle long memory latencies, and instead sensitive to memory bandwidth [93]. Wang et al [81] explored compiler analysis to identify near-optimal data placement across kernel invocations for their heterogeneous memory (mixed NVM-DRAM based) system to ensure improved power efficiency. In the context of discrete CPU-GPU heterogeneous system, Agarwal et al [3] proposed a bandwidth-aware (BW-AWARE) page placement policy that maximizes GPU throughput by balancing page placement across the memories based on the aggregate memory bandwidth available in a system. They enhanced BW-AWARE by a compiler-based profiling mechanism that furnishes programmers with data-structure access information and in turn, allows to annotate program to provide hints about memory placement. They further studied the trade-offs and considerations in relying on hardware cache-coherence mechanisms

versus using software page migration to optimize the performance of memory-intensive GPU workloads [2]. They showed that virtual address-based program locality to enable aggressive memory prefetching combined with bandwidth balancing is required to maximize performance.

However, none of the above works consider the presence of prefetcher in CPU-CPU unified memory and its impact on memory over-subscription. Because of their dense, sequential memory access, *regular* data-parallel applications benefit from prefetchers [94, 32, 70]. A prefetcher prefetches data in advance based on the spatio-temporal locality of access. In the process, it reduces the number of faults and further improves PCI-e bandwidth. Whereas, for *irregular*, data-intensive applications, aggressive prefetching can be counter-productive under memory oversubscription. The situation is aggravated further as heavily referenced pages are replaced using LRU without differentiating between cold and hot data structures. Chapter 5 introduces a user-agnostic page placement and pinning policy for data-intensive, irregular workloads in CPU-GPU heterogeneous systems. The proposed framework leverages the hardware access counter register, present in IBM Power9 [38] and Volta V100 systems, and extends the static threshold-based delayed migration strategy to employ a simple heuristic that determines a dynamic access threshold for page migration. The dynamic access threshold is derived as a response to the memory occupancy, and page access and thrashing frequency. It dynamically strikes a balance between latency-optimized direct access to remote host memory and bandwidth-optimized local memory for cold and hot data structures respectively.

### 2.4.3 Unified Framework

The memory access pattern of a GPU application depends on the fine-grained parallelism and inherent memory access characteristics. Different patterns can react differently to memory over-subscription. As a result, the oversubscription overhead of a workload heavily depends on the corresponding memory access pattern. Yu el al [92] provided a quantitative evaluation and comprehensive analysis of Unified Memory in GPUs. They profiled workload execution on a simulation platform to identify six representative classes of memory access

patterns for various general-purpose applications. Li et al [50] also classified GPU applications in three categories - 1) regular applications without data sharing, 2) regular applications with data sharing and 3) irregular applications. Burtscher et al [18] performed a quantitative study to categorize a set of irregular applications based on their memory and control flow irregularity and input dependence. Li et al [50] employed a 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, the runtime chooses between proactive eviction, memory-aware throttling, and capacity compression to address the challenge with memory over-subscription. Unlike the above works, Chapter 6 proposes an extension to the runtime for CPU-GPU interconnect-traffic pattern detection that does not rely either on intrusive profiling techniques or on any hardware extension; rather leverages prefetched block-addresses along with the access cycle information already available with the runtime engine to determine unified memory page-migration pattern and can transparently apply specific memory management techniques.

### 2.4.4 Execution of Concurrent Applications on GPU

Past studies demonstrated that sharing the GPU between multiple application kernels improves energy efficiency and overall system throughput compared to running a single kernel where on-chip GPU resources are greatly underutilized [59, 60, 76, 82, 89, 1]. Sharing GPU between multiple applications poses unique challenges as there is limited support for hardware preemption and context-switching. Existing approaches for GPU sharing can be broadly classified into two categories - (i) software-based approaches and (ii) hardware-assisted multi-tenancy. The most common software-based approach is coarse-grained time-multiplexing of kernels analogous to CPU job scheduling. Baymax [22] blocks the application, that had already consumed higher GPU time, from launching more kernels to make room kernels of other applications with lower GPU time. Because of the coarse-granularity of GPU sharing, this approach fails to improve overall resource utilization and in turn system throughput. Although the overall application lifetime may overlap, kernels from different applications run sequentially. The second software-based approach relies on code-transformation [88, 89, 59, 82].

The participating applications should be known and codes are required to be available in advance for applying any code transformation. As a result, this approach fails to facilitate the dynamic scheduling of workloads. Moreover, since kernels from different applications are fused, the hardware sees only one kernel, and in turn, may lead to unfair scheduling. Warped-slicer [89] proposed a profiling-based TB-allocation scheme for sharer kernel to improve performance. Wu et al [88] presented a transformation centered on SMs permitting precise control of job locality on SMs. Kernel Fusion [82] and Elastic Kernel [59] fuse two kernels from two different applications into one kernel based on resource requirements achieving a fine-grained sharing mechanism as they can be resident to one SM. Hardware-assisted multi-tenancy essentially leverages the hardware-based preemption and context-switching mechanism to allow sharing available compute-resources. SMs can be either physically portioned between participating kernels [76, 60] or shared partially in time [84, 85]. Wang et al [86] extended the SMK [84, 85] approach of fine-grained sharing to provide Quality of service support for collocated workloads. These approaches only considered provisioning compute resources to enable GPU multi-tenancy, and ensure performance isolation. Chapter 7 follows a kernel-to-SMs allocation scheme. Compute resources are evenly distributed among the participating concurrent application spatially.

### 2.4.5 GPU Memory Scheduling

Shortest Job First (SJF) and FR-FCFS [69, 96] are widely used as DRAM memory scheduler. Researchers also studied a DRAM scheduling policy that essentially chooses between the above two [48]. Chatterjee et al [19] explored warp-aware memory scheduling to reduce DRAM latency divergence. Jeong et al [41] presented a QoS-aware memory controller in the SoC space that allows GPUs to maintain a real-time QoS-level. These prior works on GPU memory scheduling have focused on a single execution context only to obtain the lowest possible latency without degrading the bandwidth utilization. Note that the benefits of these schedulers can also be independently adapted in the framework presented in Chapter 7 as secondary arbitration criteria. Jog et al [43] studied the application interference on on-chip interconnect from concurrent device-memory requests. Wang et al [83] presented an efficient

and fair multi-programming model in GPUs via effective bandwidth management. None of these works considered CPU-GPU interconnect as a critical resource for provisioning. With unified memory, CPU-GPU interconnect becomes a critical resource of consideration as the performance bottleneck shifts to fault-driven mitigation. Li et al [51] considered PCIe arbitration to overlap memory copies with kernel execution for multi-tenant multi-GPU systems. However, they also fail to capture the dynamic nature of applications while using unified memory stemming from memory access patterns, arithmetic intensity, and efficacy of software prefetcher. Chapter 7 focuses on CPU-GPU interconnect provisioning in the context of consolidating workloads with unified memory.

### 2.4.6 GPU Simulators

Simulation allows researchers to investigate and evaluate design ideas before implementing them on real hardware and thus drives innovation in computer architecture. GPGPU-Sim [12] is one such simulation platforms that advanced innovation in GPGPUs. It provides combined support of functional and timing modeling of GPUs based on NVIDIA's PTX ISA. However, it fails to capture interactions between the host and the device. In contrast, Multi2Sim [78] models an x86 CPU and an AMD Evergreen GPU based on AMD's GCN1 ISA. Similarly, AMD's GEM5 GPU [65] models CPU-GPU heterogeneous systems by integrating GEM5 [14], a modular full-system CPU simulator with GPGPU-Sim. MGPUSim [74] models multi-GPU systems. However, none of these simulators models unified memory runtime for discrete CPU-GPU heterogeneous memory systems.

### 2.4.7 GPGPU Workloads

Rodinia [21], Parboil [73], Lonestar [18], and PolyBench [35] are widely used benchmark suites to evaluate the performance of GPGPUs. They contain real-world computation workloads written with "copy then execute" model provided by CUDA 4.0. Recently, Chai [33] introduced 14 CPU-GPU collaborative workloads, i.e., CPU and GPU solve the working problem in tandem. Chapter 3 introduces a set mini-applications chosen from the above-mentioned benchmark suites to use CUDA unified memory APIs that are solely accelerated

by GPU to study GPU micro-architecture and software runtime without worrying about the host. Alongside, Chapter 3 introduces a set of micro-benchmarks to discover the semantics of unified memory and validate the simulated framework against real hardware. Jog et al [43] created a GPU concurrent application (GCA) framework to first evaluate the effect of bandwidth interference in concurrent applications. Chapter 3 also introduces the Unified Memory Concurrent Application (UMCA) Framework inspired by GCA.

## 3.0 UVMSmart: Simulation Framework and Unified Memory Benchmarks

High compute-density with massive thread-level parallelism of GPUs is behind their unprecedented adoption in systems ranging from data-centers to high-performance computing installations. As GPUs are increasingly being used to accelerate general-purpose applications beyond traditional graphics processing, these systems raise several design challenges, including smart runtime systems, programming libraries, and micro-architecture. However, the research community currently lacks a publicly available, comprehensive, yet highly extensible and configurable simulation framework to evaluate different design choices for GPU Unified Memory runtime. This chapter presents UVMSmart, a highly validated, functional, and timing simulator for GPU Unified Memory extending the latest branch of cycle-accurate GPGPU-Sim. UVMSmart comes with a set of micro-benchmarks, used to discover GPU runtime semantics and for performance validation along with a set of mini-applications written with CUDA Unified Memory APIs. UVMSmart introduces a large set of statistical counters along with a detailed execution timeline view enabling analysis of different architectural and runtime aspects of GPU Unified Memory.

### 3.1 UVMSmart: Design and Implementation

UVMSmart is a highly configurable and extensible GPU simulator. It is open-source. UVMSmart provides both functional and timing modeling of runtime and micro-architectural support for unified memory in CPU-GPU heterogeneous memory systems. The latest version of the GPGPU-Sim developer branch is extended to implement UVMSmart. The extended version is centered around four major components - (i) modeling micro-architectural advancement, (ii) PCIe interconnect and timing models, (iii) runtime components of unified memory, and (iv) functional support by modeling high-level programmer APIs. This section motivates the design requirements followed by a detailed discussion of the aforementioned high-level concepts.

### 3.1.1 Design Requirements

The key choices behind designing UVMSmart are summarized as follows.

1. **Modeling Function-driven Timing.** Often simulators are trace-driven and thus limit their modeling scope to timing alone. The dynamics of functional modeling and real data-driven execution guide the memory access pattern, interconnect traffic, and computation and thus results in an accurate timing model.
2. **High Configurability and Extensibility.** The goals of any good architecture simulator are - (i) to allow researchers to configure the framework based on new systems, and (ii) to ensure that it is easy to build on and extend existing research without major modifications.
3. **Accuracy of Modeling..** Simulators should be tuned and validated against real systems to serve as a valid and credible baseline. Accuracy of timing modeling and correctness of functional results should be ensured by profiling with the help of detailed micro-benchmarks and mini-applications.
4. **Ease of Use.** Architecture simulators are often difficult to set up due to complex dependencies on system and programming environments. A good simulator should be properly containerized with ease of system set-up and configuration.

### 3.1.2 Micro-architecture Modeling

**Fault-driven Migration.** Key to realizing the illusion of unified address space is micro-architectural support for fault-driven migration and on-demand allocation. Zheng et al [94] introduced the concept of replayable far-fault. A far-fault occurs when the addressed memory page is not physically present in the device memory. UVMSmart models the control flow of fault-driven migration support presented in Section 2.2.1.

**Host-pinned access and delayed migration.** UVMSmart models hardware-based page-level access counter introduced in NVIDIA Volta GPUs [71] and IBM Power9 [38] as described in Section 2.2.4. Along with access-counter threshold-based delayed migration, UVMSmart also models zero-copy, uncacheable access to host-pinned memory. If an allocation is advised to be soft-pinned to the host memory, then the memory is not copied

directly at the first-touch by the device. Rather, the migration is delayed until the number of request crosses a certain static threshold, $t_s$ configured in the model-specific register, the data is copied to the device memory. On write access, the data is migrated irrespective of the threshold [70] to maintain the exclusivity of access.

**Adpative Page Migration and Pinning.** The above discussion elucidates the standard micro-architectural features of modern GPUs. Along with these default configurations, UVMSmart also models the dynamic access threshold-based delayed page migration and pinning introduced in Chapter 5. This demonstrates how easy it is to extend UVMSmart to model newer micro-architecture and interaction between runtime and hardware counters. To implement adaptive page migration and pinning, the micro-architecture of UVMSmart is extended to maintain the access-counters per TLB entry and increment them on every access. Runtime is responsible to read and update the counters. New lower-level APIs are introduced to - (i) query access counters and keep track of access frequency in the runtime and (ii) reset them when the counters are saturated. Along with the access frequency, the runtime tracks the number of times a basic block is thrashed between the host and the device memory to determine the migration threshold under oversubscription. UVMSmart introduces a new low-level system API, invoked by the runtime to dynamically set special model-specific register and configure the dynamic migration threshold, $t_d$ for delayed migration.

### 3.1.3 Timing Model

Timing simulation for unified memory revolves around modeling different latency components associated with fault-driven migration and far-faults. UVMSmart focuses on three major components - (i) TLB look-up, shoot-down, and page-table walk, (ii) far-fault latency, and (iii) interconnect latency and various queuing delays.

**Modeling TLB and Page Table.** Pichai et al [64] proposed a `128` entry, 4 port TLB per SM. In UVMSmart, TLB is roughly modeled after theirs. The Load/Store unit per Streaming Multiprocessor (SM) is equipped with an L1 TLB and an L2 TLB shared by all SMs. UVMSmart's TLB is fully-associative which allows the TLB lookup to be done in a single-core cycle. The number of TLB entries is configurable. In general, the TLB

implementation is highly modularized and thus easy to replace. Upon capacity-miss, the TLB shoot-down is triggered invalidating entries in the LRU order. UVMSmart implements a multi-threaded GMMU page table walk [64] such that all SMs can look up for page table entries concurrently. Like TLB, page table implementation is also modularized. Thus, UVMSmart allows researchers to explore the impact of different structural choices such as associativity and set size for TLB and page table cache on overall system design. For simplicity, lookup latencies are configurable.

**Far-fault latency.** The far-fault and associated DMA requests require several PCIe round-trips and significant interaction with the host runtime as the GPU pipeline does not accommodate the capability to handle page-faults in the pipeline. The GPU memory management unit (GMMU) only walks through page-table to consolidate page faults and offloads the migration decision to the host's runtime. Far-fault latency is determined by running micro-benchmark and is defined as a parameter in the extended configuration file.

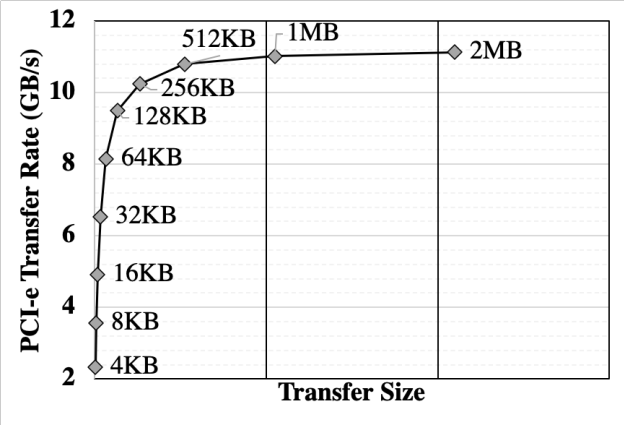



Figure 6: Transfer rate improves with transfer size as observed on real hardware with PCI-e 3.0 16x interconnect.

**PCIe latency and queuing delays.** At the heart of timing simulation is accurate modeling CPU-GPU interconnect bandwidth and transfer latency. PCI-e 3.0 16x standard implements 16 bi-directional channels with `1 GBPS` transfer speed per channel in each direction. However, due to activation overhead, the highest attainable throughput is `11GBPS`

in each direction. Interconnection bandwidth is a configurable parameter in UVMSmart. Experiments with micro-benchmarks establish a relationship between the throughput and the transfer size as shown in Figure 6. UVMSmart uses curve-fitting to derive an equation to model PCIe transfer latency. Along with transfer latency, DMA transfers also waits in several queues before being scheduled. UVMSmart also models several queuing delays suffered by a memory request generated by load/store unit in its lifetime.

### 3.1.4 Runtime Modeling

At the heart of UVMSmart is the discovery of runtime components such as software prefetchers and page eviction policies based on the detailed micro-benchmarking explained in Section 3.2.1.

**Software Prefetchers.** In unified memory, massive TLP is not sufficient to mask memory access latency as the offending warps stall for the costlier far-faults. The total kernel execution time increases dramatically and closely resembles the serialized data migration and kernel execution time of the "copy then execute" model. To ameliorate this situation, programmers resort to complicated `cudaMemPrefetchAsync` constructs to overlap the kernel execution with data migration. Software prefetchers have been proposed by researchers [94, 2] to relieve programmers from the burden of writing hand-tuned code. CUDA unified memory runtime implements a tree-based software prefetcher. UVMSmart models this software prefetcher following the detailed semantics described in Section 2.2.2.

Note that in UVMSmart, the software prefetcher is a self-contained module and thus researchers can plug-and-play any new heuristics and configure through configuration items. Along with this default software prefetcher, UVMSmart also implements ***Random*** and $64KB$ ***Sequential-local*** prefetchers [94].

- **Random ($\mathbf{R_p}$) Prefetcher.** A random prefetcher prefetches a random $4KB$ page along with the $4KB$ page for which the far-fault occurred in the current cycle. The prefetch candidate is selected randomly from the $2MB$ large page boundary to which the faulty page belongs. This not only helps CUDA workloads with random access pattern but also selecting from $2MB$ large page boundary instead of the whole virtual address space

helps in cases of the locality of memory accesses.

- **Sequential-local ($SL_p$) Prefetcher.** Zheng et al [94] described their sequential prefetcher as the process of bringing a sequence of $4KB$ pages from the lowest to the highest order of virtual address irrespective of page access pattern or far-faults. Their locality-aware prefetcher migrates consecutive 128 $4KB$ pages (or total $512KB$ memory chunk) starting from the faulty-page. UVMSmart models a different variation called sequential-local software prefetcher. Each `cudaMallocManaged` allocation is logically split into multiple $64KB$ basic blocks. The runtime, upon receiving the far-fault notification interrupts, first calculates the base addresses of the $64KB$ logical chunks to which these faulty $4KB$ pages belong. These $64KB$ basic blocks are the prefetch candidates. Further, it divides these candidate basic blocks into prefetch groups and page fault groups based on the position of the faulty page in the current basic block and then schedules them for sequential DMA transfers over CPU-GPU interconnect. Prefetching $64KB$ basic blocks ensures contiguous 16 $4KB$ pages local to the current faulty pages. Note that this is different from the locality-aware prefetcher proposed in [94]. The position of a faulty page can be anywhere within the corresponding $64KB$ basic block. Further, multiple faulty pages are taken into consideration while choosing a basic block for prefetching and can be grouped within the same $64KB$ boundary. Although, $512KB$ prefetch granularity may yield better performance compared to $64KB$ sequential local, the proposed version requires no additional coordination across multiple $2MB$ large pages.

Figure 7 compares the kernel execution time of a set of GPU workloads using different prefetching schemes against $4KB$ on-demand fault-driven migration without any prefetching. All software prefetchers improve performance significantly compared to just $4KB$ on-demand page migration under no oversubscription. This proves the necessity of prefetching in CPU-GPU unified memory. The tree-based neighborhood prefetcher provides the best performance compared to the others. This validates the adoption of such a scheme in the NVIDIA GPU driver.

Irrespective of the transfer size, every DMA transaction has a constant activation overhead from the cost of setting up the address bus. Thus, scheduling larger transfer size amortizes activation overhead and reduces transfer latency guaranteeing better bandwidth

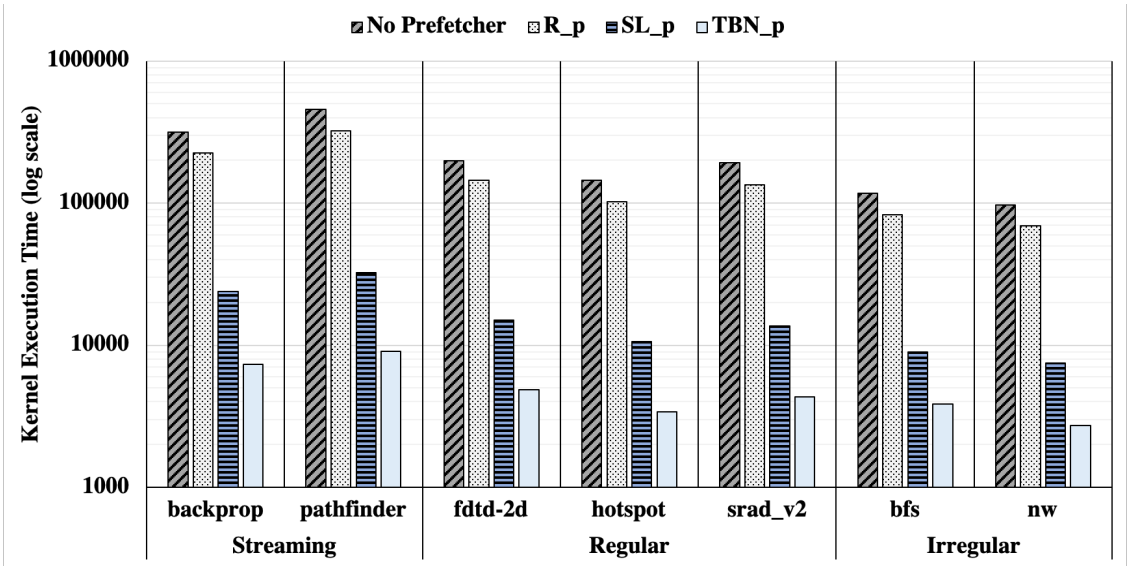



Figure 7: Comparing kernel execution time with different software prefetching schemes against no prefetching.

utilization. Both on-demand page migration and random prefetcher transfers memory in multiples of $4KB$. Whereas, $\mathrm{SL_p}$ can transfer up to $(4 + 60)KB$ memory chunks. As discussed in Section 2.2.2, $\mathrm{TBN_p}$ can migrate maximum $1MB$ of memory in a single transfer. This results in the highest bandwidth utilization and the best kernel performance achieved by the tree-based prefetcher as seen in Figure 7. Further, prefetching pages based on tree-based spatio-temporal locality reduces the number of far-faults and in turn the latency overhead of handling far-faults.

**Page Eviction Routines.** Page replacement plays a key role in unified memory by transparently dealing with device-memory oversubscription. UVMSmart implements LRU with the eviction granularity of $2MB$ x86-OS huge-page described in Section 2.2.3. Like software prefetchers, the eviction routine is modularized and self-contained. It can be easily replaced with new algorithms. Users of UVMSmart can choose from a wide-range of eviction algorithms other than the default $2MB$ LRU by simply switching between configuration options. ***Random*** and $64KB$ ***Sequential-local*** page replacements are inspired by the random

and sequential-local prefetchers [94, 32]. Along with the random and $64KB$ sequential-local eviction routines, UVMSmart also incorporates ***tree-based neighborhood pre-eviction*** introduced in Chapter 4 and ***access counter-based LFU*** presented in Chapter 5. The ability to add new eviction algorithms demonstrates the ease of extensibility of the runtime component of UVMSmart independent of any runtime API and GPU micro-architecture components.

### 3.1.5 Unified Memory API Modeling

The key to functional simulation is to support CUDA runtime APIs for unified memory such as `cudaMallocManaged`, `cudaMemPrefetchAsync`, and `cudaDeviceSynchronize` along with the traditional `cudaMemcpy` to capture the essence of different classes of CPU-GPU interconnect traffic. These APIs are compiled against `cuda_runtime_api.h` of `libcuda`. While `cudaMemcpy` serializes the execution, `cudaMemPrefetchAsync` is an asynchronous construct that uses CUDA streams to execute in parallel with kernel execution. `cudaMemPrefetchAsync` divides the total transfer in chunks of $2MB$ huge-pages to schedule DMA requests. At the core of unified memory is `cudaMallocManaged`. It provides a pointer to a virtual address space that is visible to both the host and the device. Managed allocations do not reserve memory space by default. Rather, on every page fault, memory is allocated and data is migrated keeping only one copy of data either on the host or the device. `cudaDeviceSynchronize` is a blocking call that waits for the device to complete execution before returning execution control to the host. It synchronizes page table entries between host and device. After completion of GPU execution, the host encounters page faults, and data is migrated from the device memory to the host memory.

### 3.1.6 Support for Concurrent Execution

To support Unified Memory Concurrent Application (UMCA), described in Section 3.2.3, the multi-stream execution framework of UVMSmart is leveraged. This extended framework is released publicly. The CPU thread of UMCA repeats the faster running application to overlap the execution of the slowest application in the workload composition. This is

done so that the application interference is present throughout the lifetime of participating concurrent applications. The framework captures the total execution time per application run including the time spent on far-faults and page migration. To ensure consistency, the end-to-end execution time is averaged out for repeated faster-running applications. Along with the end-to-end execution time, the simulation framework also reports multiple statistics, e.g., average weighted queuing delay, arithmetic intensity, IPC, number of conflicting network packets, and number of prioritized packets per application. This extended framework is used to evaluate the contributions in Chapter 7.

## 3.2 Application Suite

This section presents a set of micro-benchmarks to discover the semantics of unified memory runtime and validate the framework against real hardware along with a set of mini-applications written with CUDA UVM APIs. These benchmarks are used to validate the research contributions presented in Chapters 4, 5, and 6. This section also presents a suite of concurrently running applications with unified memory used to evaluate the contribution in Chapter 7.

### 3.2.1 Micro-benchmarks

Mainly four classes micro-benchmarks are designed to discover - (i) the semantics of unified memory runtime components, and (ii) timing components associated with the fault-driven migration. For sake of brevity, instead of going in detail of individual benchmarks, this section provides a high-level description of the classes of micro-benchmarks. These micro-benchmarks are publicly available along with UVMSmart.

**Managed Allocation** This class of micro-benchmarks shows that the user-provided size for a managed allocation is rounded up to the next $64 * (2^i) KB$ (e.g. $2MB192KB$ is rounded up to $2MB256KB$). These micro-benchmarks play with the size for `cudaMallocManaged` allocations and access the padded bytes safely without any memory issues. The padded ad-

dress also shows up in profiling for prefetching and page eviction. A set of micro-benchmarks in this class also identifies that the root of the tree is a maximum $2MB$ or an x86-OS huge-page. All prefetch and eviction decisions are bounded within a $2MB$ huge-page.

**Software Prefetcher** This class of micro-benchmarks is used to unravel the semantics of tree-based software prefetcher implemented by CUDA unified memory runtime described in Section 2.2.2. A single-threaded GPU kernel is used to access different indices in an array logically divide into $64KB$ chunks. Each memory access is separated by long sleep to serialize PCIe transactions. As a result, new access is made only after the previous one is completed to ensure runtime does not coalesce DMA requests. This class of micro-benchmarks proves that the minimum prefetch granularity is $64KB$ and also shows the 50% occupancy-based thresholding rule for tree-based prefetching.

**Page Eviction** This set of benchmarks helps to discover that NVIDIA GPUs implement an LRU with $2MB$ huge-page eviction granularity. They also show that no matter a huge-page is partially/fully dirty/clean, it is always written back from the device to the host memory instead of simply invalidating $4KB$ clean pages.

**Timing Components** GPU Technology Conference 2017 [71] mentions that page fault handling latency to be $30\mu s$. However, upon execution of this class of micro-benchmarks on real hardware with `GeForceGTX 1080 ti` shows it to be $45\mu s$ on average. This class of micro-benchmarks also helps find PCIe interconnect throughput and latency as a function of host-to-device memory transfer size. A single-threaded kernel transfers memory chunks of varied sizes ranging $4KB$ to $1MB$ and notes the latency and throughput for reporting.

### 3.2.2 Unified Memory Benchmarks

Out of the wide variety of mini-application suites, a set of full-benchmarks is chosen from Rodinia [21], Lonestar [18], and PolyBench [35] suites. These benchmarks are selected to ensure wide coverage of CPU-GPU PCIe access, GPU memory access pattern, and different classes of data-structures based on the frequency of access. In these benchmarks, data structures are allocated using `cudaMallocManaged` instead of `cudaMalloc`. All instances of `cudaMemcpy` API are removed, and `cudaDeviceSynchronize` is used instead to ensure safe

| Workload | Access Pattern |
|---|---|
| Streaming | 2D Convolution (2DConv),<br>Back Propagation (backprop),<br>Pathfinder,<br>StreamTriad |
| Regular | FDTD-2D,HotSpot,<br>Speckle Reducing Anisotropic Diffusion (SRAD), |
| Random | ATAX,<br>RandomAccess (RA) |
| Irregular | Needleman-Wunsch (NW),<br>Breadth First Search (bfs),<br>Single Source Shortest Path (sssp) |

Table 1: Full-benchmarks with their memory access pattern.

access of data by the host. These benchmarks are listed and classified based on their access pattern in Table 1. These access patterns are formally defined in Section 6.2.1. Along with the micro-benchmarks, these twelve mini-applications are made available.

### 3.2.3 Unified Memory Concurrent Application Framework

Unified Memory Concurrent Application (UMCA) Framework is inspired by GCA [43]. UMCA is driven by the main driver program that creates multiple CPU threads (pthreads) to launch multiple GPGPU applications in parallel. It employs mutex locks to ensure thread safety between CUDA API calls from different threads. UMCA leverages CUDA streams. A stream is defined as a series of CUDA API calls executing in-order where multiple streams can launch and execute CUDA APIs in parallel on a single GPU resource. UMCA enables concurrent execution of GPGPU applications on both real hardware and simulation frameworks without any significant code changes to the individual application source code. UMCA uses

ten applications from the unified memory application suite provided by UVMSmart [27] originally taken from Rodinia [21], Lonestar [18], and PolyBench [35] suites. These applications are shown in Figure 30. Each application is fenced with proper CUDA stream synchronization API to ensure the correctness of the output. UMCA on the simulation framework is validated for the correctness of output against real hardware. UMCA is released for further academic research.

## 3.3 Configuration and Validation

This section presents the configuration parameters for UVMSmart alongside the validation results against real hardware.

### 3.3.1 Simulation Configuration

UVMSmart extends the latest GPGPU-Sim developer branch to add functional and timing simulation support for unified memory runtime and NVIDIA GPU architecture. As part of functional support, software prefetcher, page eviction policy, and Unified Memory APIs are added and for architecture modeling, fault-driven migration and on-demand allocation, hardware access counter-based delayed migration, uncacheable access to host pinned memory is modeled. Table 2 shows the primary configuration parameters that primarily enable timing simulation for CUDA unified memory runtime on `GeForceGTX 1080 Ti` [58]. The items in boldface show the default value of the configurations items.

### 3.3.2 Containerization

UVMSmart is thoroughly containerized. Researchers are provided with a docker image that removes dependencies on the operating system, version of GPU runtime, and programming environment required to compile and execute UVMSmart. Users of UVMSmart can easily configure and switch between disparate micro-architecture choices, and prefetchers,

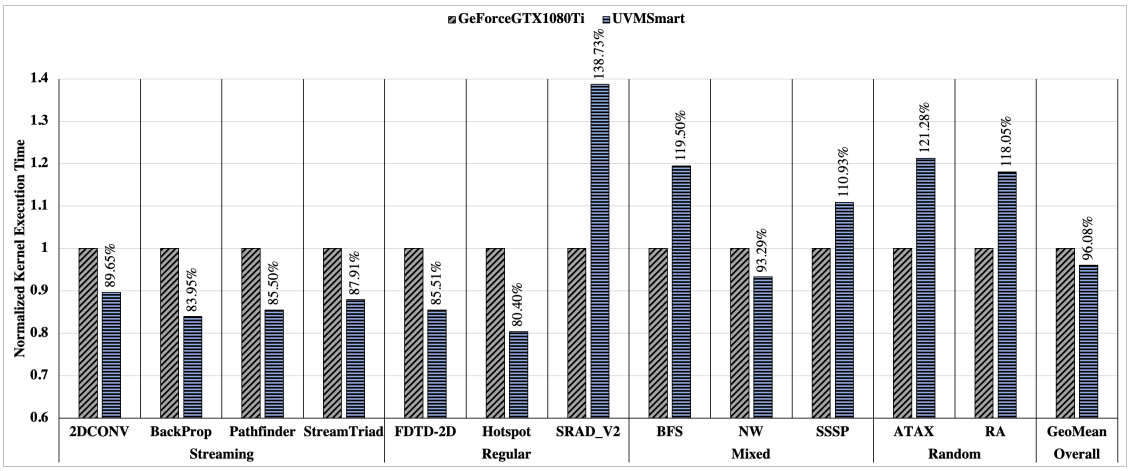
| | |
|---|---|
| **GPU Cores** | 28 SMs, 128 cores each @ 1481 MHz |
| **Shader Core Config** | Max. 32 CTA and 64 warps per SM,<br>32 threads per warp, GTO scheduler |
| **Page Size** | 4KB |
| **Page Table Walk Latency** | Multi-threaded, 100 core cycle |
| **TLB** | Fully-associative L1 |
| **CPU-GPU Interconnect** | PCI-e 3.0 16x,<br>8 GTPS per channel per direction,<br>100 GPU core cycles latency |
| **DRAM Latency** | 100 GPU core cycles [3] |
| **Remote Zero-copy Access Latency** | 200 GPU core cycles |
| **Eviction Granularity** | **2 MB**, 64KB |
| **Page Replacement Policy** | **LRU**, LFU, Sequential 64KB, Random |
| **Far-fault Handling Latency** | $45\mu s$ |
| **Software Prefetcher** | **Tree-based**, Sequential 64KB, Random |
| **Static Access Counter Threshold** | **8** |
| **Multiplicative Migration Penalty** | **8** |

Table 2: Configuration parameters of the simulated system.

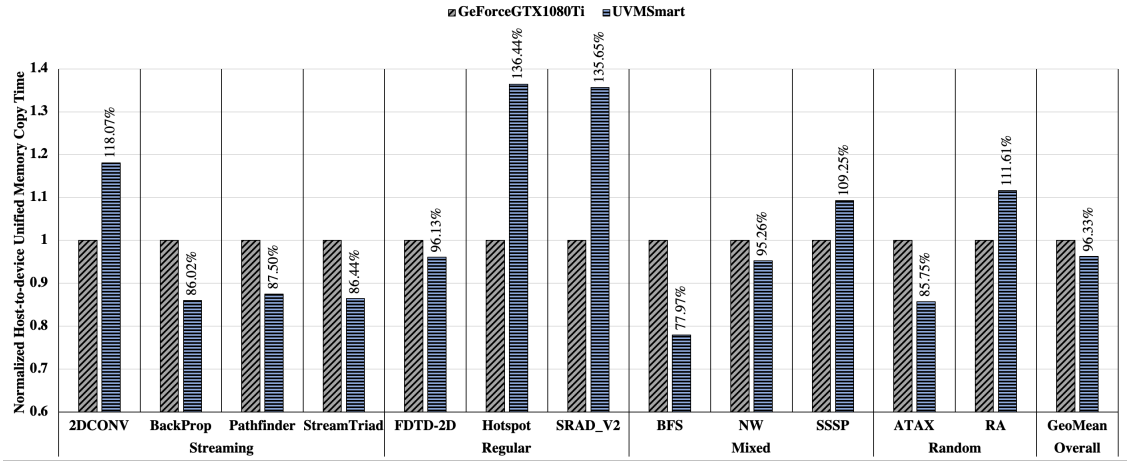
eviction algorithms, and memory management techniques by updating items in the extended configuration file without modifying the simulator and/or any programming environment.

### 3.3.3 Validation

This section reports the results of validating UVMSmart against real hardware. This comparison builds confidence in the correctness and accuracy of functional and timing simulation respectively. Figure 8 shows the validation results. Figure 8a compares the estimated execution time by UVMSMart and the real hardware execution time of benchmark kernels. Across all the benchmarks, the difference between these two values is 4% (geometric mean) and ranging between 4% to 38%. Similarly, Figure 8b compares the total host-to-device memory copy time between UVMSmart and real hardware with (geometric) mean variation




(a) Execution Time




(b) Host-to-device Memory Copy

Figure 8: Performance validation of UVMSmart against real hardware.

as 4% and ranging between 4% to 36%. Note that the original GPGPU-Sim already has an average 15% performance deviation in comparison to the real hardware setup for benchmarks not using unified memory.

# 4.0 Adaptive Page Replacement

Section 3.1.4 has demonstrated that good software prefetcher is the key to the success of CPU-GPU Unified Virtual Memory. Both $SL_p$ and $TBN_p$ migrate memory in the multiples of $64KB$ basic block local to the current faulty pages. This is with the hope that the thread blocks will eventually access these pages in the immediate future. However, in reality, some of these pages may not be referenced before the eviction procedure starts replacing pages. These unused prefetched pages are never chosen for eviction by LRU. Instead when GPU memory capacity has been reached and kernel execution stalls for new page migration, a heavily referenced page could be chosen for displacement. Thus, an eviction policy, unaware of prefetchers, meets with the challenge of how to deal with the memory oversubscription issue. A logical choice would be evicting pages in the same way they were brought in by the hardware prefetchers. This means pre-evicting pages in multiples of $64KB$ basic blocks based on sequential or tree-based neighborhood locality. Locality-based pre-eviction has two benefits. ① Evicting pages in larger chunks increases PCI-e write-back bandwidth and lowers the write-back latency. ② Software prefetchers can work in tandem with the pre-eviction scheme. This also means that it overcomes the drawbacks of memory threshold-based pre-eviction policy. This chapter introduces two new pre-eviction schemes.

## 4.1 Pre-eviction Policies Adaptive to Prefetchers

### 4.1.1 Sequential-local ($SL_e$) Pre-eviction

Sequential-local eviction consults the LRU page list to select an eviction candidate. The runtime then determines the $64KB$ basic block to which the current eviction candidate belongs and then schedules the whole basic block for eviction and eventual write-back. Note that there can be pages in the basic block which were not accessed and just brought in by the prefetcher. All the 16 pages in the $64KB$ are written back as a single unit irrespective

of the pages within are clean or dirty. This is because transferring memory in larger chunks improves PCI-e bandwidth and reduces latency instead of writing back multiple $4KB$ pages.

### 4.1.2 Tree-based Neighborhood ($TBN_e$) Pre-eviction

The proposed tree-based neighborhood pre-eviction strategy is inspired by the $TBN_p$. It leverages the full-binary tree structures created and maintained for hardware prefetching at the time of managed allocation. Thus, it accounts for no additional implementation overhead. As discussed in Section 2.2.2, all nodes in these full-binary trees correspond to $64KB$ basic blocks and the root node of each tree corresponds to a maximum contiguous virtual space of $2MB$ large page or a size equivalent to $2^i * 64KB$. Like $SL_e$, an eviction candidate is chosen from the LRU list. Then a $64KB$ basic block, to which this eviction candidate belongs, is identified for pre-eviction. After the selection of every pre-eviction candidate, the runtime traverses the whole tree updating the valid page size of all its parent nodes including the root node by subtracting the size of the evicted basic block. At any point, if the total valid size of any node is strictly less than 50% of the maximum valid size of that node, a further pre-eviction decision is made by the runtime which is in turn pushed down to the children till the leaf level. This process continues recursively till no more basic blocks can be identified for pre-eviction or no node higher than leaf-level including the root node has valid size less than 50% of the maximum capacity at the corresponding tree level. The eviction granularity in this scheme varies between $64KB$ to $1MB$ and thus it adapts between the two extremities of $4KB$ and $2MB$ eviction granularity for LRU.

Figure 9 demonstrates the $TBN_e$ on a $512KB$ memory allocation for simplicity. Initially, let us assume that all pages in this $512KB$ allocation are valid in the page table. Let us further assume that the first three entries in the LRU list correspond to the basic blocks $N_0^1$, $N_0^3$, and $N_0^4$. Upon over-subscription, when page replacement routine kicks in, these three basic blocks are identified for eviction one after another. After evicting the first three basic blocks, the valid size for each of the nodes $N_1^0$, $N_1^1$, and $N_1^2$ is updated to $64KB$ by the runtime. Further the valid sizes for nodes $N_2^0$, $N_2^1$, and $N_3^0$ are updated as $128KB$, $192KB$, and $320KB$ respectively. Let us now assume that the current least recently used

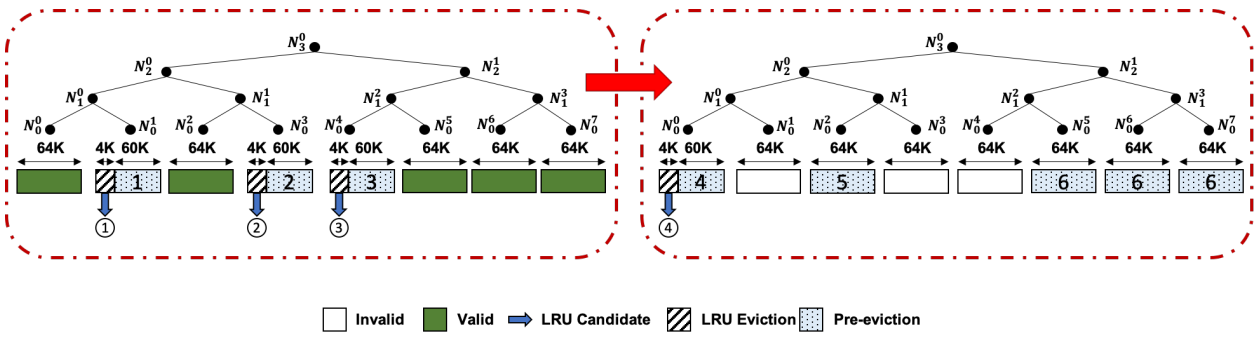



Figure 9: Demonstration of tree-based pre-eviction on 512 KB memory chunk.

page corresponds to the basic block $N_0^0$. After the fourth pre-eviction, the valid sizes of $N_1^0$, and $N_2^0$ are updated as $0KB$ and $64KB$ respectively. As the current valid size for $N_2^0$ is $64KB$ and is less than 50% of its maximum capacity, a further pre-eviction decision is made for $N_1^1$ and is pushed to its children. This ultimately chooses $N_0^2$ as a pre-eviction candidate. At this point, the runtime traverses the tree and updates the valid sizes of all nodes in the tree. It then discovers the valid size of $N_3^0$ to be $192KB$ which is less than 50% of its maximum capacity. This pushes the pre-eviction decision to $N_2^1$ and in turn to its children. This process identifies basic blocks $N_0^5$, $N_0^6$, and $N_0^7$ as pre-eviction candidates. As these blocks are contiguous, the runtime groups them together into a single transfer.

### 4.1.3 Specific Design Choices

Both $\text{SL}_\text{e}$ and $\text{TBN}_\text{e}$ first select an eviction candidate from the LRU list and then identify the corresponding $64KB$ basic block for eviction. These basic blocks, up for eviction, can have some pages with dirty and/or access flags set in the page table along with some pages for which these flags are not set and only valid bits are set in the page table. We make a distinct design choice for how the LRU page list is to be maintained in case of these pre-eviction policies. We place all the pages in the LRU when the valid flags of the corresponding page table entries are set in the GPU page table. This means the LRU list contains all pages with the valid flag set in the GPU page table in contrast to the traditional LRU list which

only maintains pages with the access flags set in the page table. Further, a page is pushed to the back of the LRU list upon any read or write access in the course of execution. Upon evicting a basic block, all pages including the eviction candidate are removed from the LRU list. Hence, this design choice ensures all pages local to the eviction candidate are evicted irrespective of whether they are accessed or not. This is how $\mathrm{SL_e}$ and $\mathrm{TBN_e}$ deal with the unused prefetched pages migrated by the $\mathrm{SL_p}$ and $\mathrm{TBN_p}$ and free up contiguous virtual address space. We sort the pages first at a large page level based on the access timestamp of the $2MB$ chunk they belong to. Then, within the $2MB$ large page, $64KB$ basic blocks are sorted based on their respective access timestamps. This hierarchical sorting ensures a global order at $2MB$ large page level and the local order of $64KB$ basic blocks at the leaf-level of $2MB$ tree.

A known issue with LRU is that the performance degrades for a repetitive linear access pattern. For example, if there are $N$ pages in the LRU page list, a CUDA kernel executing a loop over an array of $N+1$ pages will face a far-fault on every access. There have been a lot of research efforts invested in the past in modifying LRU to work with repetitive sequential access pattern as iterating over large arrays are common. One such proposal is to switch to Most Recently Used (MRU) page replacement policy upon detecting such a memory access pattern. However, detecting or predicting memory access patterns in runtime is itself a challenging problem and incurs large implementation and performance overheads. In this paper, we follow a simple solution to address this problem by reserving certain pages from the top of the LRU page list such that they are not chosen as eviction candidates. Thus, reserving the top percentage of the LRU page list reduces thrashing since the top percentage of pages in the LRU list, which are chosen for immediate eviction, are also accessed first in the next iteration.

## 4.2 Experimental Evaluation

### 4.2.1 Pre-eviction Policies in Isolation

Section 3.1.4 has shown that the $TBN_p$ has the best performance when device memory can accommodate the whole working set. So, experiments in this chapter only consider $TBN_p$ before over-subscription. Under over-subscription, the simulator disables prefetcher and only migrates $4KB$ pages on-demand. The reason behind this setup is to investigate the sole impact of different eviction policies on the kernel execution time. Also for this experiment, working sets for the benchmarks are set as 110% of the device memory size. Figure 10 shows the result of this experiment.

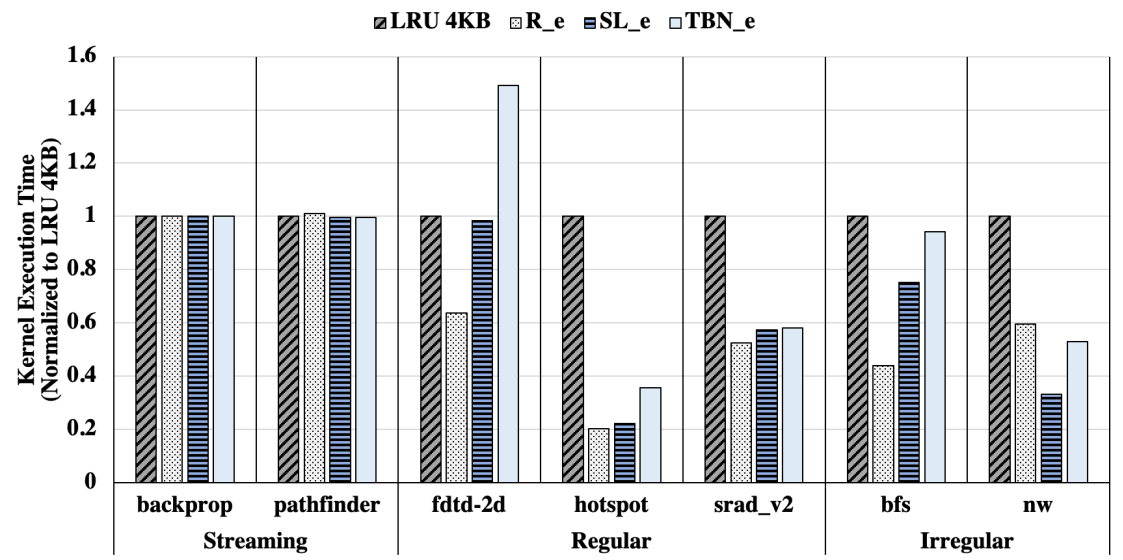



Figure 10: Comparing the effect of different eviction policies on kernel execution time. Tree-based prefetcher is active before reaching device memory capacity. Upon over-subscription, software prefetcher is disabled and $4KB$ pages are migrated on-demand. Working set is 110% of the device memory size.

The following major behaviors are exhibited by the benchmarks. ① `backprop` and `pathfinder` show no sensitivity to the choice of eviction policy. This is because both of

these benchmarks exhibit a streaming memory access pattern. Both of them scan a large vector in parts sequentially and do not reuse data across different iterations. ② For most benchmarks, random eviction policy provides the best performance contrary to the popular belief [94] that LRU and random page replacement policies have no performance difference. Randomly picking a $4KB$ eviction candidate from the entire virtual address space reduces the chance of thrashing. In contrast, the following LRU list increases thrashing for iterative kernels with data reuse.

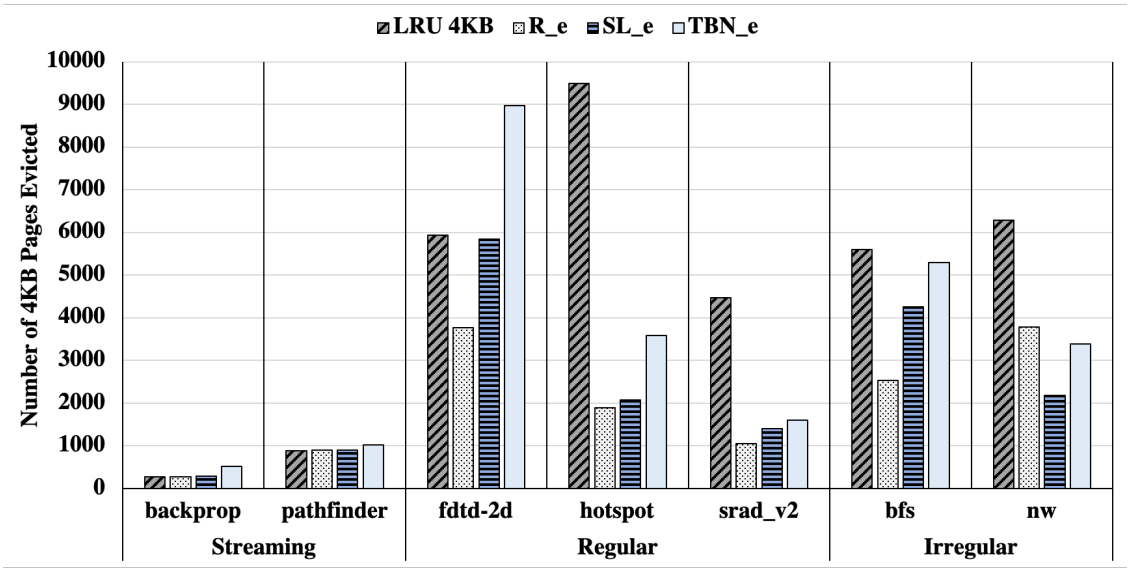



Figure 11: Comparing total number of pages evicted for different eviction schemes.

Figure 11 reports the total number of $4KB$ pages evicted by the different schemes. It can be observed that the kernel performance is highly correlated to the total number of pages being evicted by the corresponding page replacement policy as expected.

### 4.2.2 Combinations of Pre-eviction Policy and Software Prefetcher

To this end, this experiment takes the logical next step by pairing eviction policies and hardware prefetchers under over-subscription. The experiment enables the $\text{TBN}_\text{p}$ before over-subscription. Also, 110% device memory oversubscription is considered. Four different

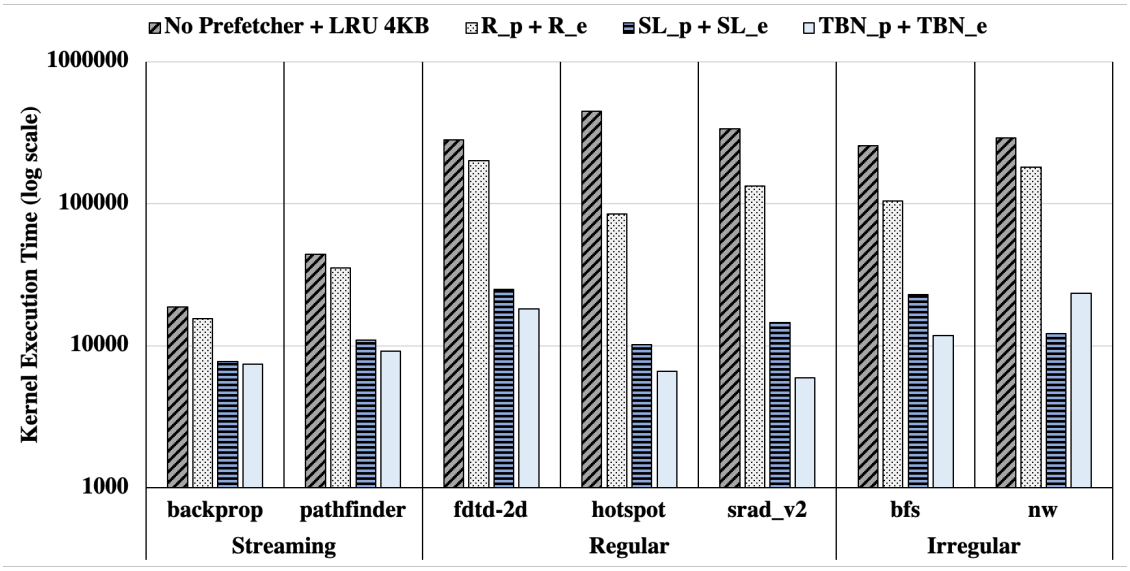
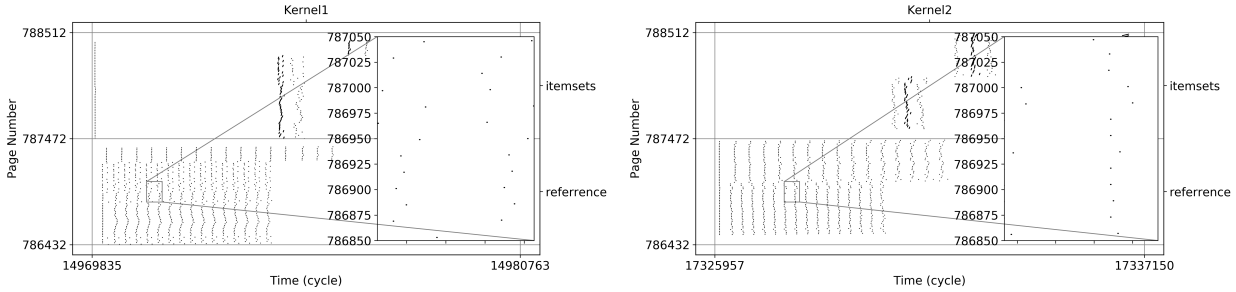



Figure 12: Comparing the effect of different combinations of eviction policies and software prefetcher after oversubscription on kernel execution time. Tree-based prefetcher is active before reaching device memory capacity. Working set is 110% of the device memory size.

combinations of eviction policies and page migration schemes are chosen in a way that they do not violate, rather respect each other's semantics.

Figure 12 reports the kernel execution time for these settings under over-subscription: (i) LRU $4KB$ eviction and no hardware prefetching, (ii) $R_e$ policy and $R_p$, (iii) $SL_e$ and $SL_p$, and (iv) $TBN_e$ and $TBN_p$. Note that the third and fourth combinations drastically outperform the first two. In particular, the combination of $TBN_e$ and $TBN_p$ provides an average 93% performance improvement compared to the combination of LRU $4KB$ eviction policy and $4KB$ on-demand page migration. This can be attributed to the higher PCI-e read and write bandwidth utilization achieved by these combinations as they evict and prefetch memory in larger granularity other than $4KB$ which is the case for the first two combinations. Further, pre-eviction reduces the page access time by not waiting for pages to be written back, and allowing prefetchers to prefetch pages reduces the number of page faults.

One exception is `nw`. The combination of $SL_e$ and $SL_p$ yields better performance com-




(a) Iteration 60

(b) Iteration 70

Figure 13: Page access pattern of nw benchmark without eviction.

pared to the combination of $\text{TBN}_\text{e}$ and $\text{TBN}_\text{p}$. To gain more insight into this behavior, let us further analyze the memory access pattern of `nw`. In this example, `nw` runs for 127 iterations. Figure 13 shows the pages being accessed in iterations 60 and 70 (chosen randomly) respectively. The horizontal axis corresponds to the core cycle and the vertical axis shows the virtual page number. As it can be seen that for `nw`, in every cycle, a set of pages, which are spaced far apart in the virtual address space, are accessed repeatedly over time. As the memory access is sparse yet localized and repeated over time, the smaller granularity of eviction yields better performance than larger granularity. This is because evicting pages in a larger chunk by $\text{TBN}_\text{e}$ causes more thrashing than evicting $64KB$ basic blocks by $\text{SL}_\text{e}$.

### 4.2.3 Memory Over-subscription Sensitivity

In this experiment, the percentage of memory oversubscription is varied to study the scalability of the combination of the proposed pre-eviction policy and software prefetcher. The combination of $\text{TBN}_\text{e}$ and $\text{TBN}_\text{p}$ is used after over-subscription for this experimental setup as this combination outperforms other combinations in general as seen in the previous section. Figure 14 shows that `backprop`, and `pathfinder` shows no sensitivity to memory over-subscription percentage as they exhibit streaming memory pattern. Other than `nw`, all other benchmarks scale up linearly. The order of magnitude performance degradation with

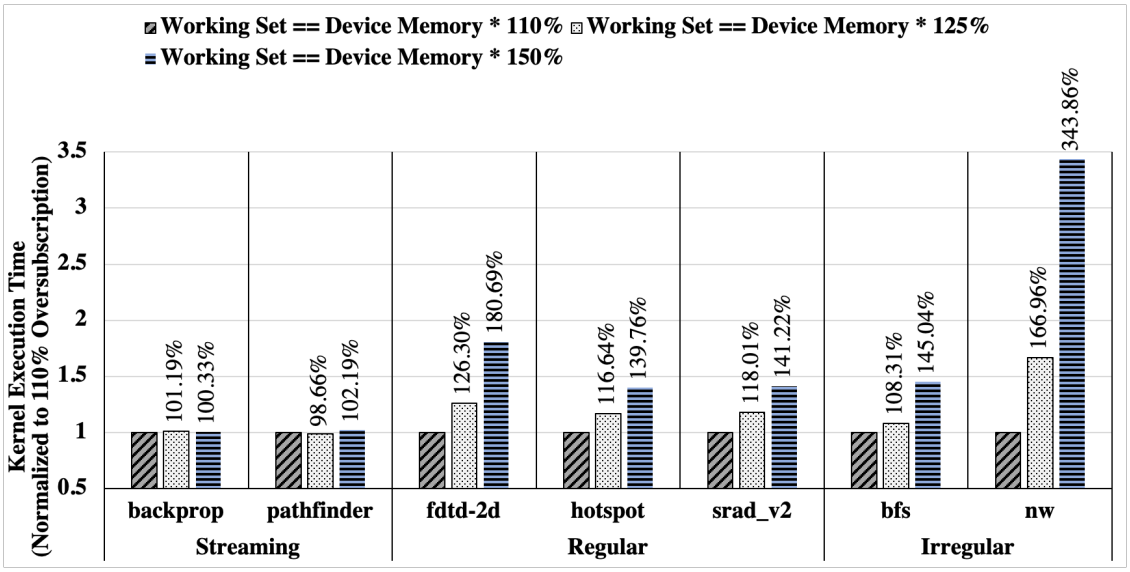



Figure 14: Sensitivity of combinations of tree-based prefetcher and pre-eviction to the percentage of memory over-subscription by the working sets.

a higher percentage of memory over-subscription for `nw` can be attributed to its localized sparse memory access and large thrashing caused by the same.

### 4.2.4 Reserving Percentage of LRU Page List from Eviction

To address the issue of page thrashing for benchmarks with data reuse over multiple iterations, a certain percentage of pages from the top of the LRU page list is reserved from eviction as discussed in Section 4.1.3.

Figure 15 compares the kernel execution time of the benchmarks with the 10% and 20% reservation of the LRU page list along with the combination of $TBN_e$ and $TBN_p$ against the same with no reservation. The streaming applications like `backprop` and `pathfinder` show no performance variation with the LRU page reservation. The kernel performance improves with 10% reservation from the top of the LRU list for all other benchmarks. However, with higher percentage of reservation, it hurts for certain benchmarks.

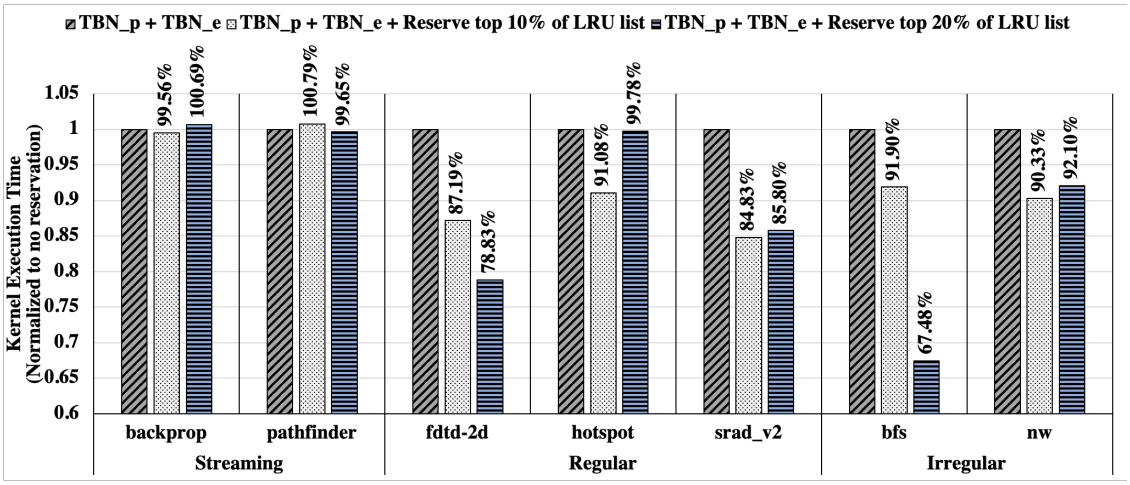



Figure 15: Effect of reserving a certain percentage of pages of LRU list from eviction on kernel runtime. Working set is 110% of the device memory size. Tree-based prefetcher is active before reaching device memory capacity.

### 4.2.5 2MB Large Page Eviction

Based on the experimental results presented in the previous sections, it can be safely concluded that pre-evicting pages in larger granularity based on the spatio-temporal locality within $2MB$ large page enables further hardware prefetching under oversubscription and in turn provides better performance. Further, $4KB$ LRU eviction renders hardware prefetching ineffective. So, a question can be asked then "Why not replacing pages in $2MB$ granularity?". Evicting $2MB$ large pages means invalidating the entire tree. This ultimately guarantees contiguous invalid pages required for the software prefetcher to work. However, like aggressive prefetching, aggressive eviction is detrimental as it can cause serious page thrashing upon evicting highly referenced pages in case of repetitive kernel launch.

In this experiment, $\text{TBN}_\text{e}$ is compared against the static $2MB$ LRU. Figure 16 shows that the $\text{TBN}_\text{e}$ ensures an average 18.5% and up to 52% performance improvement compared to $2MB$ LRU under 110% memory over-subscription. By opportunistically determining a dynamic replacement granularity based on the current state of the $2MB$ full-tree, $\text{TBN}_\text{e}$ nav-

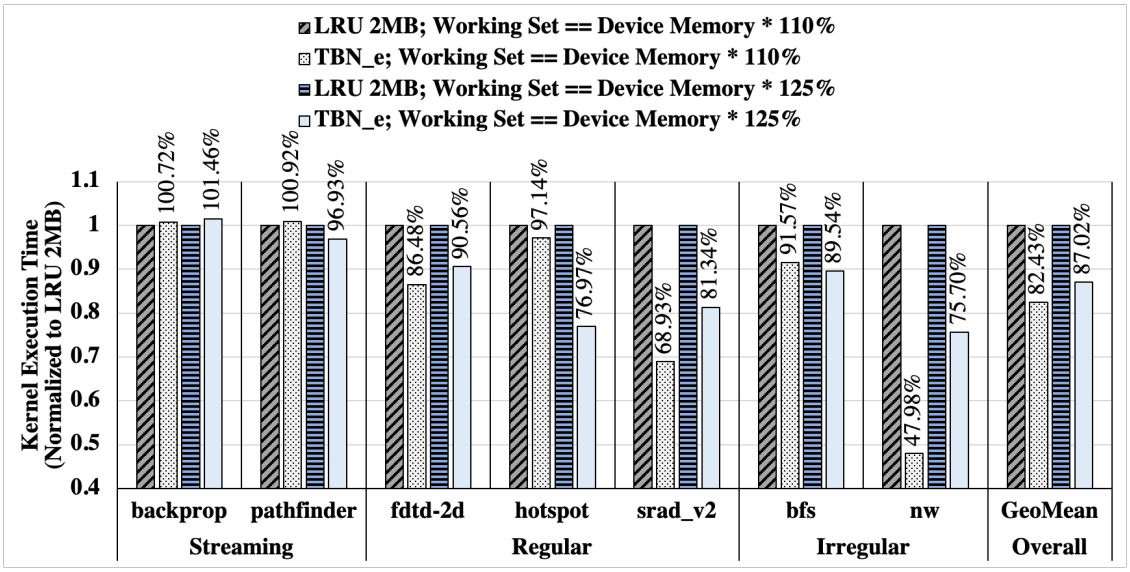



Figure 16: Comparing the performance of tree-based pre-eviction against 2MB large page eviction.

igates between the spectrum of $4KB$ and $2MB$ LRU eviction and overcome the limitations with both of these two extremes.

Figure 17 shows the average page thrashing caused by $2MB$ large-page eviction and $\text{TBN}_\text{e}$ under 110% and 125% memory over-subscription. As usual, `backprop` and `pathfinder` shows no thrashing as they do not have any data reuse. For benchmarks like `bfs`, `hotspot`, `nw`, and `srad` the performance improvement by $\text{TBN}_\text{e}$ compared to $2MB$ eviction can be attributed to the significant reduction in the number of page thrashing.

## 4.3 Conclusions

This chapter introduces locality-aware pre-eviction policies that are compatible with software prefetcher. Memory access patterns of the unified memory workloads have been studied in depth to gain more insight into the interplay of such pre-eviction policies and

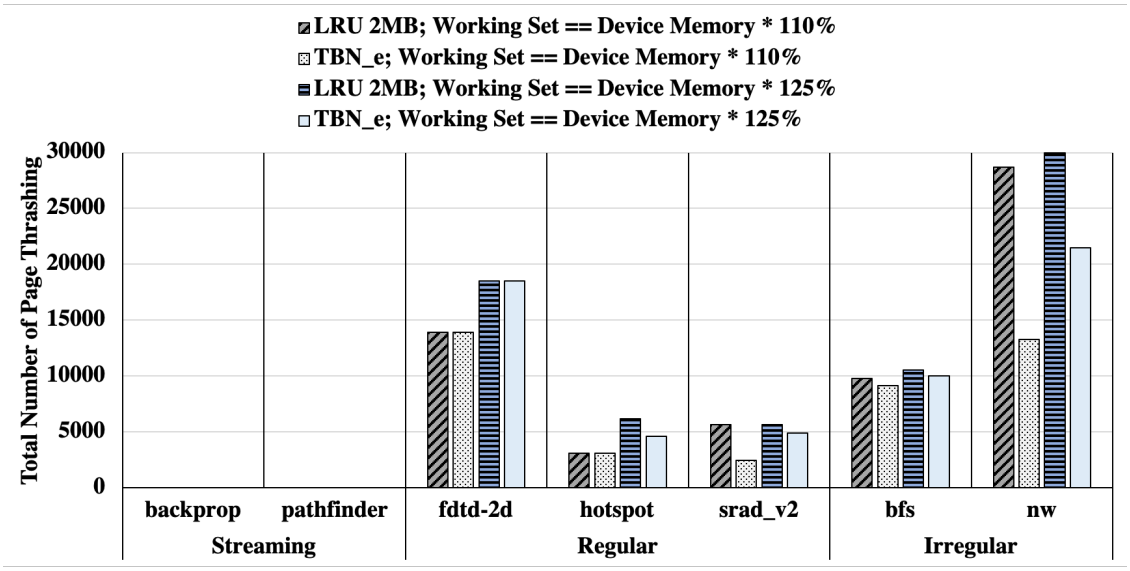



Figure 17: Comparing the effect of tree-based pre-eviction and 2MB large page eviction on the total number of pages thrashed.

software prefetchers. Experimental results demonstrate that the proposed tree-based pre-eviction policy provides an average of 93% and 18.5% performance speed-up compared to LRU based $4KB$ and $2MB$ page replacement strategies, respectively. The proposed scheme moves between two extremes of $4KB$ and $2MB$. By opportunistically determining a dynamic eviction size based on spatio-temporal locality within $2MB$ large page, it overcomes the limitations with page replacement strategies with fixed granularity. Moreover, as these pre-eviction schemes leverage the existing tree-based implementation of software prefetcher, they do not cost any additional implementation overhead. This makes this solution simple, pragmatic, and adaptable on real hardware irrespective of vendor-specific architectures.

# 5.0 Adaptive Page Migration and Pinning

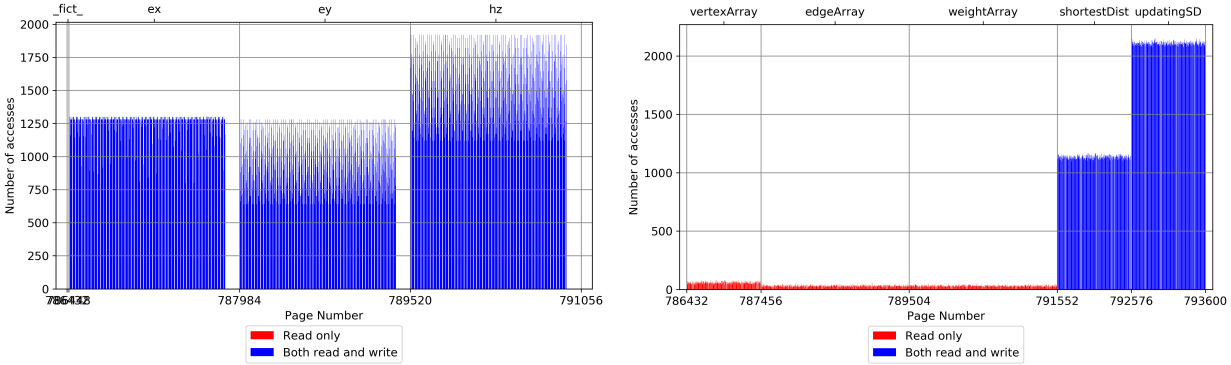
The performance overhead under memory oversubscription depends on the memory access pattern of the corresponding workload. While a regular application with sequential, dense memory access suffers from long latency write-backs, the performance of an irregular application with sparse, seldom accesses to large data-sets degrades due to page thrashing. Although smart spatio-temporal prefetching and large page eviction yield good performance in general, remote zero-copy access to host-pinned memory proves to be beneficial for irregular, data-intensive applications. Further, new generation GPUs introduced hardware access counters to delay page migration and reduce memory thrashing. However, the responsibility of deciding what strategy is the best fit for a given application relies heavily on the programmer based on a thorough understanding of the memory access pattern through intrusive profiling. This chapter proposes a programmer-agnostic runtime that leverages the hardware access counters to automatically categorize memory allocations based on the access pattern and frequency. The proposed heuristic adaptively navigates between remote zero-copy access to host-pinned memory and first-touch page migration based on the trade-off between low latency remote access and high-bandwidth local access.

## 5.1 Motivation

An effective memory management strategy to deal with device memory oversubscription requires a thorough understanding of the memory access pattern of the workloads. To this end, the memory access patterns of various GPGPU workloads are analyzed in detail to characterize their respective behaviour. The workloads are broadly categorized into: 1) *regular* with dense, sequential, repetitive memory access and 2) *irregular* with sparse, seldom access.

### 5.1.1 Workload Characterization

Firstly, the distribution of page access frequency of different memory-allocations/data-structures is visualized over the entire execution period of two benchmarks, `fdtd` and `sssp`, in Figure 18. Memory pages are classified based on the type of access - *read only*, and *both read and in-place write*.



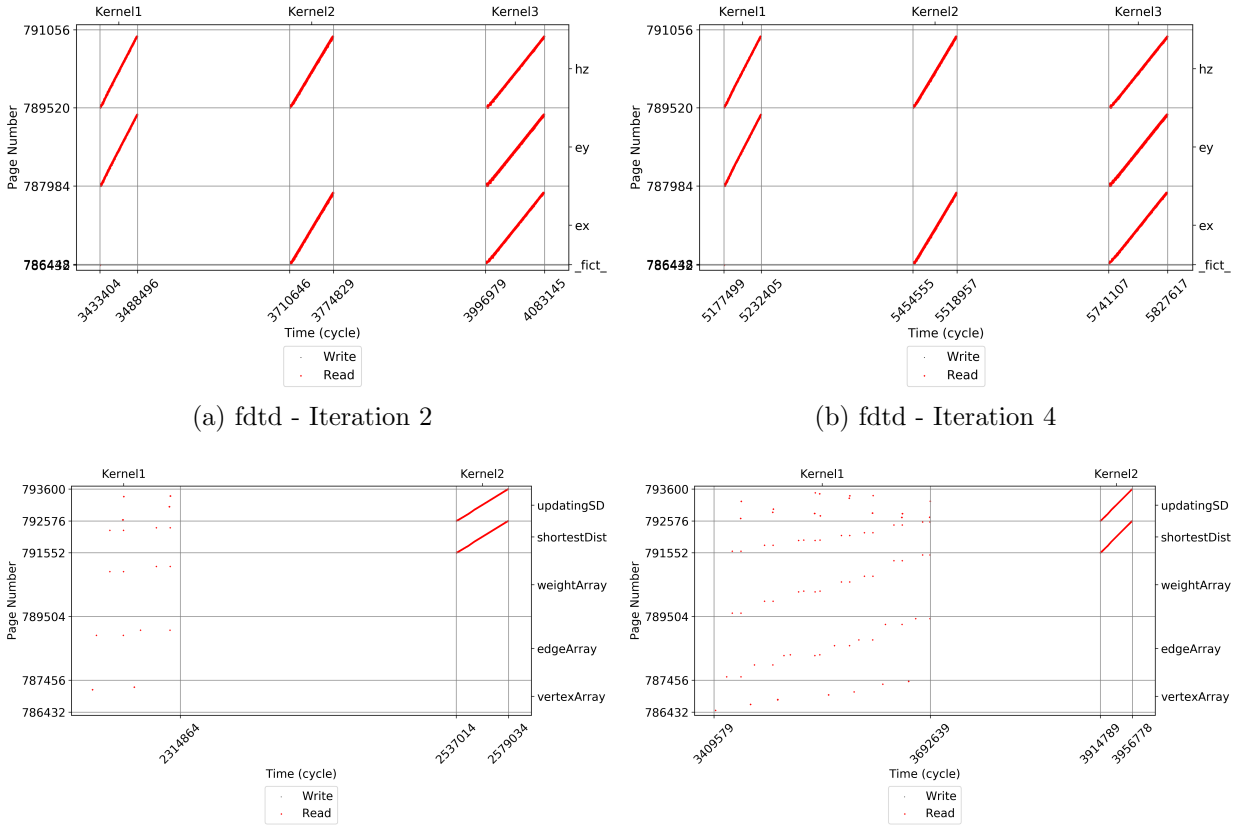

(a) fdtd (b) sssp

Figure 18: Visualizing page access distribution detailing type of access and total number of accesses per page per managed allocation for fdtd and sssp.

Figure 18a shows that in `fdtd`, most of the pages in the allocations are accessed at the same frequency over the entire execution time. A very few pages, equally spaced over the allocation boundary, are accessed a lot more than the rest of the pages. On the other hand, Figure 18b shows an entirely different characteristics for `sssp`. Note that few allocations/data-structures are more heavily accessed than the others leading to a cluster of *hot* and *cold* pages over the entire memory set. Moreover, the read-only data-structures are *cold* and the pages in *hot* data-structures are both read from and written to. This shows that for *irregular* applications a small fraction of memory footprint corresponds to the higher share of bandwidth.

Figure 19a, and 19b show the memory access pattern of `fdtd` in iterations 2, and 4 respectively. The horizontal axis represents time in cycles and the primary vertical axis shows the page numbers accessed at a given cycle. Note that the memory access pattern is

fairly constant over two different iterations. Moreover, in every iteration, each allocated data structure is accessed linearly. This explains the access frequency distribution of `fdtd` reported in Figure 18a. Thus, `fdtd` is characterized as a *regular* application. Regular applications typically show dense, sequential access repeated over multiple iterations. `backprop`, `hotspot`, and `srad` are other examples of GPU benchmarks that can also be categorized as *regular* applications as they exhibit similar memory access pattern [21].




(a) fdtd - Iteration 2

(b) fdtd - Iteration 4

(c) sssp - Iteration 3

(d) sssp - Iteration 5

Figure 19: Visualizing page access patterns of a regular (fdtd) and an irregular (sssp) application over two iterations. (a), and (b) show access pattern of fdtd in iterations 2, and 4 respectively. (c), and (d) show access pattern of sssp in iterations 3, and 5 respectively.

On the other hand, Figure 19c, and 19d show the memory access pattern of `sssp` in iterations 3, and 5 respectively. Note that `kernel1` exhibits sparse memory access over

different allocations/data-structures and the memory pages accessed over different iterations vary drastically in virtual address space. However, `kernel2` shows sequential and dense access over two data structures in every iteration. This justifies the cluster of *hot* and *cold* data structures in `sssp` as shown in Figure 18b. Hence, `sssp` is characterized as an irregular applications. In general, *irregular* applications exhibit dense sequential access on *hot* data structures and sparse, random access on *cold* data structures. `bfs`, `nw` [21], `ra` [79] are other benchmarks that fall under the same category.

### 5.1.2 High-level Observations

Based on the discussion in the previous section, the following observations can be made to motivate the proposed work.

1. A higher percentage of interconnect bandwidth is consumed by a small percentage of the total memory pages,
2. Migrating pages of *cold* data-structures/allocations causes eviction of pages of *hot* data-structures/allocations for *irregular* applications,
3. Data migrations due to page thrashing over low bandwidth interconnect contribute to memory oversubscription overhead,
4. Current state-of-the-art solutions are not satisfactory to all workloads as zero-copy access and delayed migration can hurt the performance of regular applications although proven to be useful for irregular workloads,
5. Thus, an effective solution to address device memory oversubscription must rely on user-hints based on an extensive understanding of memory usage and access pattern.

## 5.2 Dynamic Access counter Threshold Based Delayed Data Migration

This section introduces an adaptive runtime heuristic that is programmer-agnostic as it requires no advice to the memory subsystem from the application developer. Further, the extended runtime leverages the new hardware features of page-level access counters. Thus,

it demands no hardware modification and is solely based on a pragmatic modification to the GPU runtime.

### 5.2.1 Dynamic Access Counter Threshold

In current delayed migration solutions, pages are always migrated only after crossing a static access counter threshold. This means regular applications, with dense memory access, end up incurring the overhead of remote memory access before ultimately migrating the pages to the local memory. Moreover, when there is no memory constraint, it is always beneficial to migrate the data to the device memory and access it locally. This is because the tree-based prefetcher can considerably improve PCI-e bandwidth utilization and in turn reduce the number of far-faults. Also, local memory is bandwidth optimized and thus guarantees better performance than fragmented remote access.

So, an effective solution should be able to decide how to eliminate the overhead of remote access for no memory oversubscription and regular applications in general. This section proposes a dynamic threshold for delayed migration. The heuristic of the proposed solution is driven by the Equation 1.

$$t_d = \begin{cases} t_s * \frac{\text{Num. of allocated pages}}{\text{Total num. of pages}} + 1, & \text{if no oversubscription} \\ t_s * (r+1) * p, & \text{otherwise} \end{cases} \tag{1}$$

where $t_s$ = Static access counter threshold,

$r$ = Number of round trips or number of times evicted,

$p$ = Multiplicative Migration Penalty

The proposed dynamic threshold, $t_d$, grows adaptively in response to the size of free space in the device memory starting from 1 to the driver configured static threshold. Firstly, assume that the static threshold, $t_s$, is configured in the driver as 8. If currently less than 12.5% of device memory is allocated, then the dynamic threshold is derived as 1 from Equation 1. This means every first touch will cause page migration. Similarly, the dynamic access counter threshold will be the same as the static threshold of 8 just before reaching the full capacity of device memory and 9 upon oversubscription. The goal of the framework, here, is

to tame down the aggression of the prefetcher by delaying the page migration as the memory starts filling up to its maximum capacity. Use cases involving no memory oversubscription and regular applications benefit from this mechanism compared to delayed migration based on a static threshold.

Equation 1 also addresses the situations involving memory oversubscription. The framework is driven by the intuition that under memory oversubscription *cold* pages should be *soft-pinned* to the host memory and the only *hot* pages should be copied to the device memory. This is because *hot* pages can benefit from bandwidth-optimized local-memory access and the sparse and seldom access to *cold* pages can benefit from low-latency of remote-access without contributing to the strict local memory budget. Equation 1 also introduces a multiplicative penalty for migration under oversubscription, $p$ configurable as a module parameter to the GPU driver. With $p = 2$ and $t_s = 8$, the pages are migrated after $16^{th}$ access after oversubscription. This helps reduce the amount of page thrashing. Moreover, the framework keeps count of the number of round trips or the number of times a certain chunk of memory is evicted which is denoted as $r$ in Equation 1. For example, if a given chunk of memory is evicted twice, then the dynamic threshold of migration for that memory chunk will be derived as 48. The intuition behind this heuristic is that the more a page is thrashed, the harder it should be pinned to the host memory. Thus, the heuristic controls the hardness (/softness) of page pinning and helps achieve the concept of host-pinned zero-copy allocation for highly thrashed memory pages.

### 5.2.2 Access Counter Based Page Replacement

The framework also extends the page replacement strategy leveraging the same access counters. As detailed in Section 2.2.3, a naïve LRU page replacement cannot differentiate a set of *cold* pages from a set of *hot* pages. As a result, it may end up evicting highly referenced *hot* pages in the process of migrating a *cold* page and thus defeats the objective of hard-pinning hot pages to the device memory and cold pages to the host memory. Instead, the access counters are used to sort the list of $2MB$ large pages in the LRU list such that cold pages are prioritized over hot pages for eviction in the irregular applications. Thus,

it emulates a simplified Least Frequently Used (LFU) scheme in the framework. However, with linear sequential access in regular applications, where pages are accessed with almost the same frequency, the framework automatically falls back to the LRU policy. Read-only pages are also prioritized as eviction candidates. This is because, on write access, pages are migrated exclusively to the device memory irrespective of their access counter. So, the framework prefers to keep the write pages in the device-local memory as much as possible.

### 5.2.3 Implementation Details

**Access Counter Granularity.** Access counters are maintained at the page granularity for Volta GPUs [71]. However, as explained in Section 2.2.2, the tree-based prefetcher in `nvidia-uvm` module migrates data in multiple of $64KB$ basic blocks based on the page faults relayed from GMMU. This leads to the optimization of maintaining access counters at $64KB$ basic block level instead of $4KB$ page granularity. This not only reduces the memory overhead of maintaining access counters, but it is also functionally more meaningful as the prefetch granularity is $64KB$.

**Access Counter Maintenance.** The implementation uses $32bits$ access registers. Hardware counters are updated by GMMU on every page access during the TLB lookup. Whereas, runtime reads the values of hardware access counters and maintains them as part of the driver (/system software) memory. As runtime is responsible to update GPU's page table, they are read, updated, and consulted on every PCIe migration. The lower $27bits$ are used for access counters and most significant $5bits$ are kept to keep track of round trip time or $r$. This provides the opportunity to maintain a large value for access frequency and to realize a historic counter. The access counters in Volta GPUs only keep track of remote accesses. In comparison, the framework maintains the count of both device-local and remote accesses. This provides a historic view of accesses and differentiates hot pages from cold pages over larger iterations. When the counter for one of the basic blocks reaches the maximum value (for either the round trip counters or the access counters), the framework halves the corresponding counters of all the basic blocks instead of resetting them entirely. This helps maintain the relative view of hotness over multiple allocations.

## 5.3 Experimental Evaluation

Henceforth, the dynamic access-counter threshold based delayed migration scheme will be alternatively referred as *Adaptive*. The *Adaptive* scheme is compared with ① the state of the art baseline where remote access is not enabled and data is migrated at first touch, alternately referred it as *Baseline* or *Disabled*, ② the static access counter based threshold proposed in Volta GPUs termed as *Always*, and ③ a static access counter based delayed migration scheme enabled only after oversubscription referred as *Oversub*. Difference between *Always* and *Oversub* is that *Always* delays migration from the start irrespective of memory oversubscription.

For *Baseline*, LRU page replacement is active whereas, for the other three schemes, the proposed access counter-based simplified LFU policy is used. Note that the following experiments only deal with 125% of device memory oversubscription. Unlike, CPU virtual memory, current GPUs are not capable of handling a higher percentage of memory oversubscriptions. NVIDIA recommends using more than one GPU to distribute workload if the GPU memory oversubscription is more than 125%.

### 5.3.1 Sensitivity to Static Migration Threshold

The success of access counter-based delayed migration relies on finding a suitable value for the static access counter threshold, $t_s$. The objective of the framework is not to hurt *regular* applications in general and all applications under no oversubscription. This experiment aims to find out the sensitivity of $t_s$ on the kernel execution time. Figure 20 shows the result. In this experiment considers the *Always* scheme as it is the state of the art for delayed migration.

As $t_s$ is varied from 8 to 32, *regular* applications show almost no sensitivity to the static threshold. This is because for regular applications the number of per basic block accesses generated by load/store unit is quite high and they always exceeds the threshold. Thus, for *regular* application, no remote access is performed. However, *irregular* applications shows sensitivity to $t_s$. While the performance of `nw` and `sssp` degrades with higher value of $t_s$,

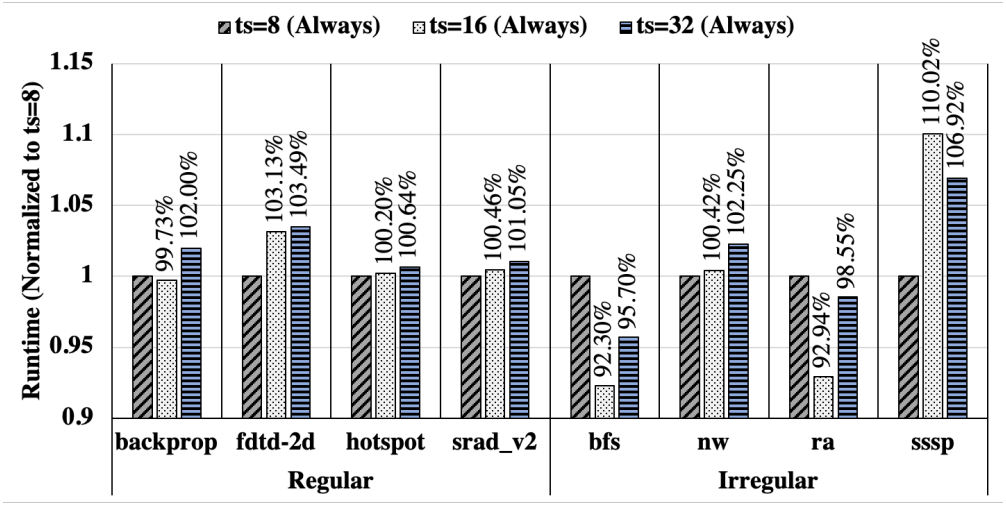



Figure 20: Sensitivity of workloads to the static access counter threshold for delayed migration.

`bfs` and `ra` show improvement for $t_s = 16$ compared to $t_s = 8$. This behaviour is not unpredictable and depends on the input of the workload and the sparsity of memory access. This work recommends a justifiably small number for $t_s$ such that it closely resembles first-touch migration under no oversubscription. However, an extremely small value for $t_s$ like 1 or 2 is also not recommended as it will hurt performance for irregular applications under oversubscription. Experiments in the later subsections use $t_s = 8$. Section 5.3.4 will present experimental results showing the sensitivity $p$ on performance.

### 5.3.2 The Case of No Oversubscription

This section compares the proposed *Adaptive* scheme against the *Baseline* and the *Always* scheme for delayed migration under no oversubscription. Figure 21 shows the normalized runtime of different workloads using the above three schemes. *Oversub* is not applicable for this experiment as it enables threshold-based delayed migration only after oversubscription.

Figure 21 shows for both *regular* and *irregular* applications, the *Adaptive* scheme produces results equivalent to the *Baseline* or *Disabled* scheme. This means that the dynamic

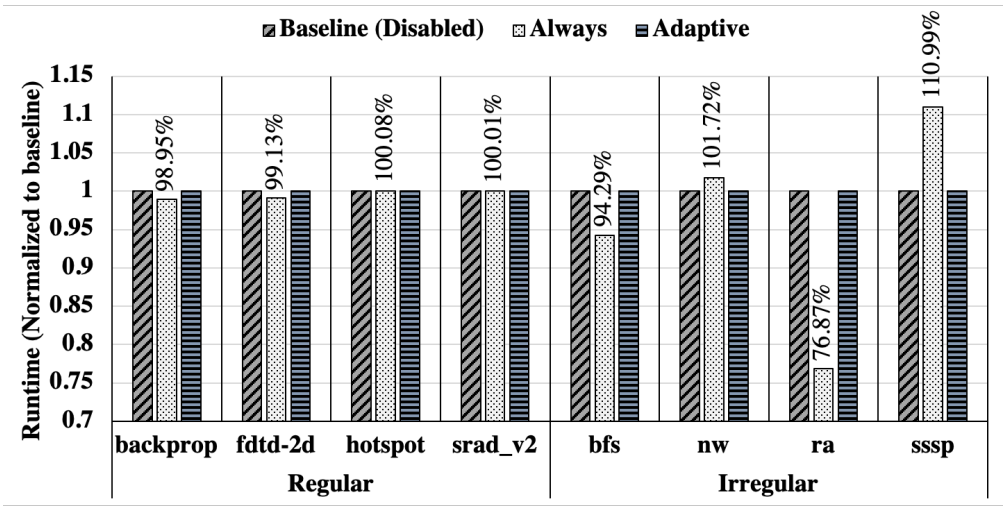



Figure 21: Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration scheme under no memory oversubscription.

threshold scheme falls back to first-touch migration based on the access frequency and memory availability. While, for *regular* applications, *Always* scheme shows no major performance difference, for *irregular* applications, it introduces unpredictability. `bfs` and `ra` benefit from *Always* scheme, whereas `nw` and `sssp` show performance degradation. This is because even for sparse memory access if there is no memory constraint, it is always better to copy the data to the device memory using a prefetcher and then benefit from bandwidth optimized local access. Note, that the objective of the framework under no oversubscription is not to outperform *Baseline* first touch-based migration, rather show more consistent and predictable behavior compared to the *Always* scheme of static threshold-based delayed migration.

### 5.3.3 The Case of Oversubscription

This experiment demonstrates the effectiveness of the proposed *Adaptive* policy by comparing runtime of different workloads against *Disabled*, *Always*, and *Oversub* policies. For the *Adaptive* scheme $p = 8$ is used and $t_s = 8$ is set for all delayed migration policies.

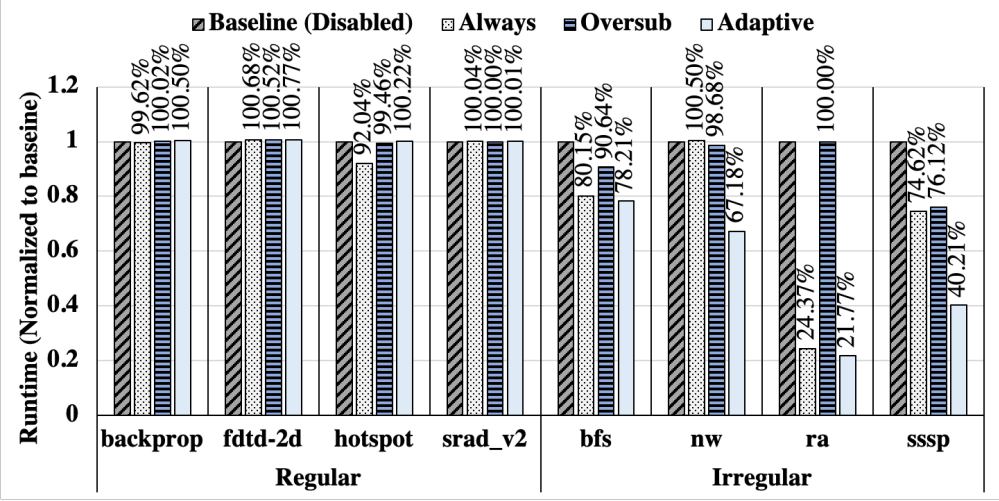



Figure 22: Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes.

Figure 22 shows that *Adaptive* scheme does not impact performance of *regular* applications. On the other hand improves the performance of *irregular* applications by 22% to 78%. Moreover, it also yields better performance compared to static access counter threshold based schemes.

To reason about the performance improvement by the *Adaptive* scheme demonstrated in Figure 22, Figure 23 reports the number of pages being thrashed for different schemes. As it can be seen that the improvement in kernel execution time is directly a factor of reduction in memory thrashing for *irregular* applications. For *regular* applications, the number of pages being thrashed using *Adaptive* scheme is same as *Baseline* or *Disabled*. Note that for `backprop` there is no thrashing at all. This is because it scans through the entire allocation sequentially without any data reuse over iterations. On the other hand, `ra` shows completely random access and no data reuse which makes it a perfect candidate for zero-copy host-pinned memory access.

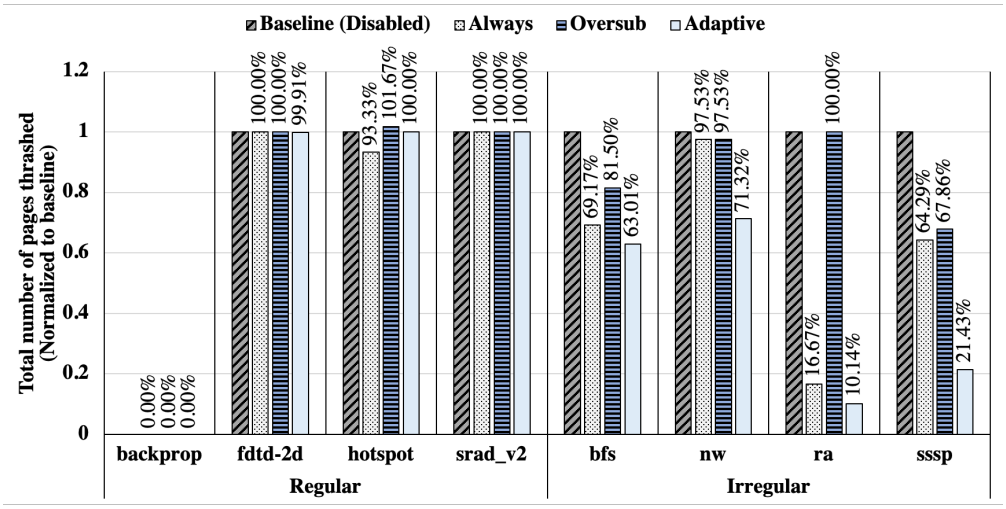



Figure 23: Comparing the impact of dynamic access counter based adaptive scheme on memory thrashing against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes.

### 5.3.4 Sensitivity to Multiplicative Penalty

This experiment is aimed to study the effect of the multiplicative penalty, $p$ on the kernel execution time. The intuition is that higher values of $p$ dictates a larger dynamic threshold for delayed migration and thus achieves harder pinning of pages.

Figure 24 shows that *regular* application doesn't show any performance variation when the value of $p$ is varied from 2 to 8. Whereas, *irregular* applications shows strictly linear performance improvement with larger $p$. The observation is consistent with $p = 16$ and $p = 32$ (not plotted due to limited space). This is how the adaptive scheme navigates between bandwidth optimized local access and low latency remote access.

A question may arise as to why not having an unreasonably higher value of $p$. Clearly, the dynamic threshold, $t_d$ is dictated by $p$. Hence, for relatively large $p$, the values of $t_s$ and $r$ would not have an appreciable effect on $t_d$. As a result, an unreasonably large $p$ will blindly keep pages pinned to the host memory without caring for the access threshold or the number of round trips (evictions). Note that $t = 1048576$ indeed has huge performance

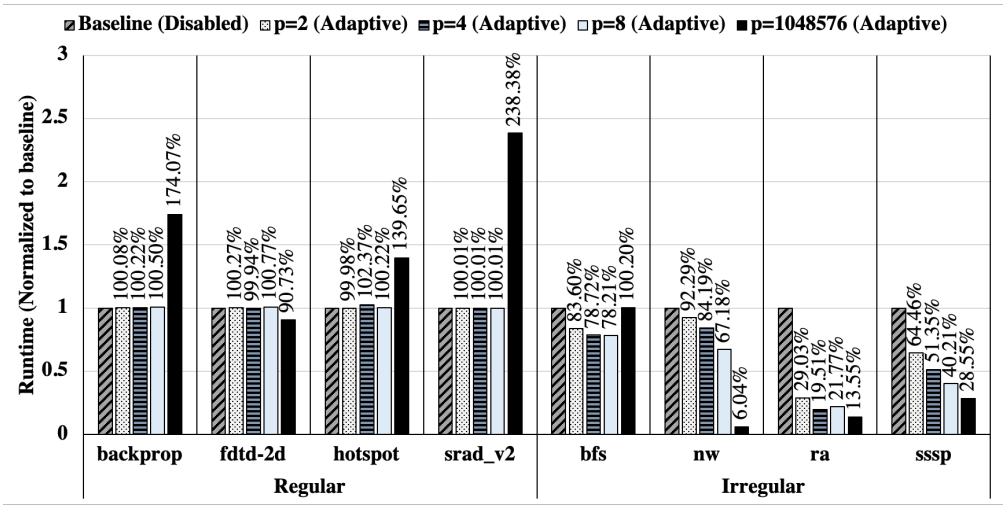



Figure 24: Sensitivity of workloads to the multiplicative migration penalty.

benefits on `nw`, `ra`, and `sssp` by eliminating thrashing entirely. However, this behaviour is unpredictable and solely depends on what pages get pinned to the host memory. For example, `bfs` shows 2% performance degradation. Moreover, regular applications suffer a great deal of performance loss for a large $p$. For example, the kernel execution time for `srad` almost doubles up. This is because for dense, sequential access it is always better to migrate the memory to the device and access locally. This also proves that the framework is tunable to achieve remote zero-copy access by configuring $p$ or multiplicative penalty. Further, the dynamic threshold based heuristic navigates between zero-copy remote access and first touch migration adaptively.

Note that the sensitivity studies in Section 5.3.1 and 5.3.4 are not performed to find the optimal values for $t_s$ and $p$, rather to show the effectiveness of the heuristic for reasonable values for these two parameters. Moreover, the objective of the framework is not to automate the process of finding values for $t_s$ and $p$ as these are configurable as kernel module parameters to NVIDIA driver.

### 5.3.5 Access Counter Based Eviction

This section shows the effectiveness of the access counter-based simple LFU in comparison with the LRU page replacement implemented in NVIDIA GPUs. The experimental setup involving both eviction policies uses *Adaptive* scheme with $t_s = 8$ and $p = 8$. The working sets of the workloads are set at 125% of the total device memory size.

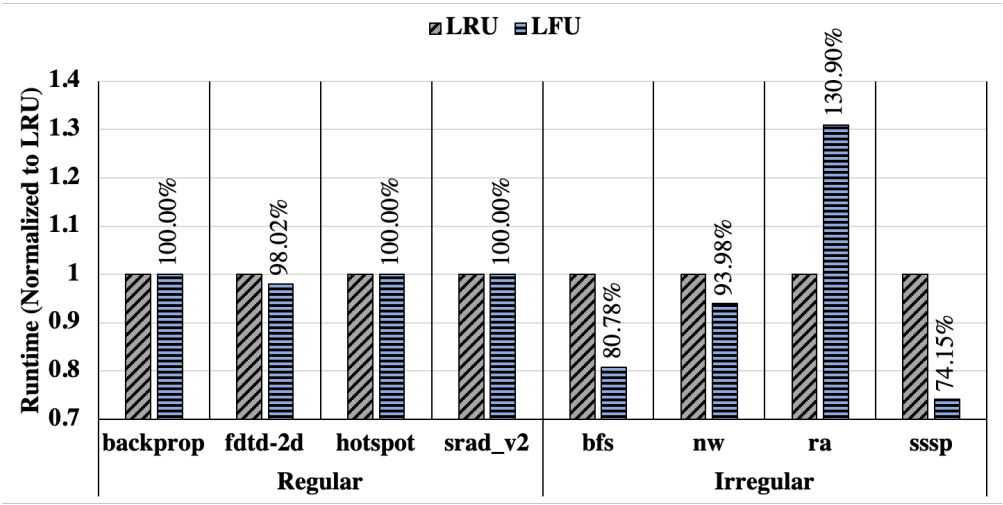



Figure 25: Performance variation between LRU and LFU page replacement strategies.

Figure 25 shows that for *regular* applications the choice of page replacement policy has no impact on the performance. *Regular* applications access the data sequentially and access counters at $2MB$ level per managed allocation are almost the same. Hence, LFU converges to LRU. On the other hand, LFU improves the performance of *irregular* applications by 6% to 26%. This validates the hypothesis that prioritizing cold pages over hot pages for eviction improves performance. However, `ra` shows a sharp performance degradation of 30% using LFU. This can be contributed to its extremely sparse and random access.

### 5.3.6 Invalidating Clean Pages

As discussed in Section 2.2.3, the eviction granularity is $2MB$ for NVIDIA GPUs. Irrespective of the pages being dirty or clean, $2MB$ large pages are written back from the

device to the host memory. As seen in Figure 18b, irregular workloads like `sssp` has a high percentage of memory allocation which is read-only. Such an application can benefit from invalidating clean pages instead of waiting for always writing back $2MB$ pages. Simple optimization can be performed to keep track of access type at $2MB$ large page level and based on the type decide on whether to write back or invalidate. This means even if a single page is dirty within $2MB$, the whole chunk is considered dirty. This is done to reduce the overhead of going over all $4KB$ pages within $2MB$ checking for dirty or clean status. Moreover, transfer bandwidth for $2MB$ is much higher and guarantees lower latency. This experiment compares the always write back scheme with the opportunistic mix of writing back dirty pages and invalidating clean pages. For both schemes, *Adaptive* scheme with $t_s = 8$ and $p = 8$ under 125% memory oversubscription is considered.

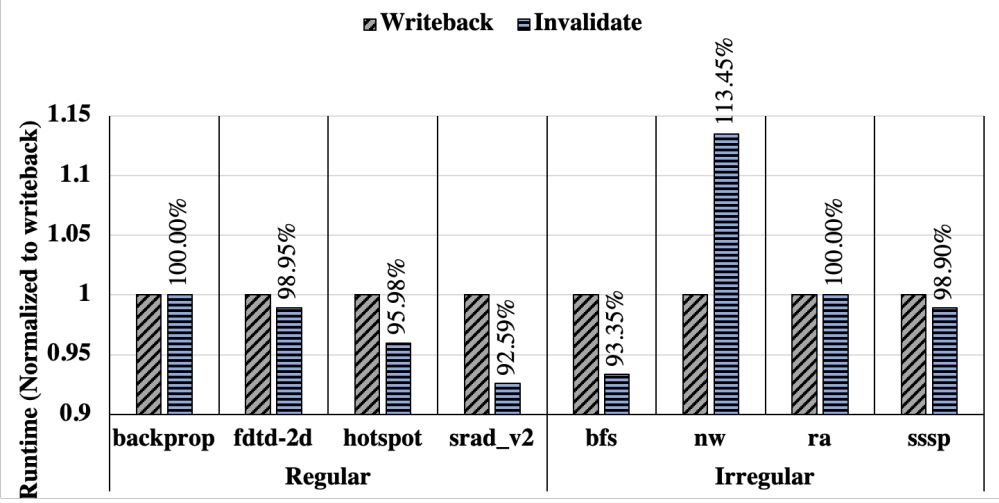



Figure 26: Comparing performance of schemes where (i) $2MB$ blocks are always written back and (ii) only dirty pages are written back and clean pages are invalidated directly.

Figure 26 shows that invalidating read-only memory directly can improve performance by 2% to 7% compared to the always writing back strategy. However, *nw* shows a sharp performance degradation of 13%. This is because when any page from a $2MB$ chunk, which is staged for write-back, is addressed by load/store unit, the $2MB$ chunk is removed from the staging queue of write-back and put back to the end of the LRU page list. As $2MB$

blocks scheduled for eviction suffers from queuing delay, it may get the opportunity to be removed from the queue and brought back to the list of valid pages. However, invalidating clean pages is almost instantaneous and thus the blocks being invalidated do not get an opportunity to remain valid in the memory from any access in the immediate future. This is the case with `nw`. As a result, invalidating clean pages increases thrashing for `nw` and thus degrades performance. Whereas, `backprop` and `ra` show no performance difference as they have no reusable data.

## 5.4 Conclusion

This chapter introduces a programmer-agnostic framework to deal with memory oversubscription overhead stemming from page thrashing in irregular, data-intensive GPU applications. The adaptive scheme leverages the hardware access counters present in new-generation GPUs. Hence, it makes the solution simple and pragmatic with no need for any programmer-assistance or new hardware enhancements. Based on the memory availability and access frequency, the heuristic adaptively navigates between first-touch page migration and remote zero-copy access. The proposed framework employs a dynamic access counter threshold to delay page migration instead of relying on a static threshold for accesses. Based on access frequency, the proposed scheme achieves the soft-pinning of hot pages to the device local memory while remotely accessing cold pages from host memory. As a result, it balances between low latency remote access and high bandwidth local access to reduce thrashing significantly. Experimental results show that while the proposed framework improves performance for irregular applications under a tight memory budget, it has no negative impact on performance in cases of no memory oversubscription or for regular applications.

# 6.0 An Adaptive Unified Framework

The CUDA APIs and unified memory runtime offer multiple memory management primitives to optimize the performance of a wider range of applications. However, the onus of selecting the best memory management strategy falls squarely on application developers. As no single solution can address all disparate memory management requirements presented by various workloads, application developers typically resort to intrusive profiling to characterize workloads and the allocations within. This chapter presents a smart adaptive runtime that simplifies memory management for application developers and effectively addresses the performance overhead under device memory oversubscription. This extended runtime introduces three components - (i) a pattern detection engine, (ii) a policy engine transparent to the application programmer, and (iii) an augmented adaptive memory management module. The adaptive runtime chooses from a wide array of memory management policies namely dynamic page migration and pinning, tree-based eviction, uncacheable host-pinned access to reduce thrashing of unified memory pages under device memory oversubscription.

## 6.1 Motivation

The memory access pattern of a GPU application depends on the fine-grained parallelism and inherent memory access characteristics. Different patterns can react differently to memory over-subscription. As a result, the oversubscription overhead of a workload heavily depends on the corresponding memory access pattern. Yu el al [91] provided a quantitative evaluation and comprehensive analysis of Unified Memory in GPUs. They profiled workload execution on a simulation platform to identify six representative classes of memory access patterns for various general-purpose applications. Li et al [50] also classified GPU applications in three categories - 1) regular applications without data sharing, 2) regular applications with data sharing and 3) irregular applications. They employed a counter in each SM's load/store unit to sample the number of coalesced memory accesses and deter-

mine the memory access pattern of the executing workload. Upon detecting the memory access pattern, the runtime chooses between proactive eviction, memory-aware throttling, and capacity compression to address the challenge with memory oversubscription.

Unlike the above works, this chapter looks into page migration patterns. Page migration patterns are inherently different from memory access patterns. For example, a particular workload can migrate the pages in the device memory during cold start and access it locally across different kernel launches over multiple iterations. If there is no oversubscription and subsequent page thrashing pattern of accessing the device-local memory after cold start has little to no effect on page thrashing and performance overhead. Page migration is a compounded behavior of - (i) memory requirements of scheduled warps, (ii) efficacy of coalescing unit and MSHRs, and (iii) most importantly the heuristic employed by software prefetcher. Under oversubscription, access patterns and eviction heuristic also indirectly influence page migration decisions and 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.

## 6.2 The Unified Framework

The chapter proposes an extension to the unified memory runtime for page migration pattern detection that does neither rely on intrusive profiling techniques nor any hardware extension. Rather, it leverages the information of $64KB$ basic blocks identified by prefetcher for fault-driven migration to detect page migration patterns. Then, based on the detection result, the smart framework adaptively chooses and applies suitable memory management policies to deal with memory oversubscription and reduce interconnect memory-traffic.

### 6.2.1 Memory Migration Pattern

The following are the page migration patterns considered and modeled to realize the adaptive smart runtime extension.

**Regular.** Equation 2 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.

$$(p_1, p_2, ..., p_k)^N \text{ where, } N > 1, k > \text{Memory Size} \tag{2}$$

**Streaming.** A streaming migration pattern is a special case of the *Regular* access pattern where no data is re-referenced. With $N = 1$, Equation 2 is transformed into Equation 3, which represents a streaming migration pattern. This pattern has no locality in its references or in other words, memory pages have an infinite re-reference interval.

$$(p_1, p_2, ..., p_k) \text{ where, } k = \text{Number of Accesses} \tag{3}$$

**Random.** Equation 4 defines a *Random* migration pattern where basic-blocks, ranging between $[1, m]$, has a migration probability of $\varepsilon$. Further, $\varepsilon$ is high, close to 1 for the common case. This means that memory chunks are migrated randomly based on some probability distribution.

$$P_\varepsilon(p_1, p_2, ..., p_m) \text{ where, } m = \text{Number of Accesses} \tag{4}$$

**Irregular/Mixed.** Equation 5 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 5 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, $\varepsilon$. This entire access can be repeated for $N$ iterations. Note that unlike *Random* migration pattern, the probability of migration, $\varepsilon$ is almost close to 0 in the common case.

$$((q_1, q_2, ..., q_k)^M P_\varepsilon(p_1, p_2, ..., p_m))^N \tag{5}$$

### 6.2.2 Pattern Detection

The main contribution of this chapter is the pattern detection engine augmented to the smart adaptive runtime. The heuristic employed by the pattern detection engine is illustrated by the following steps.

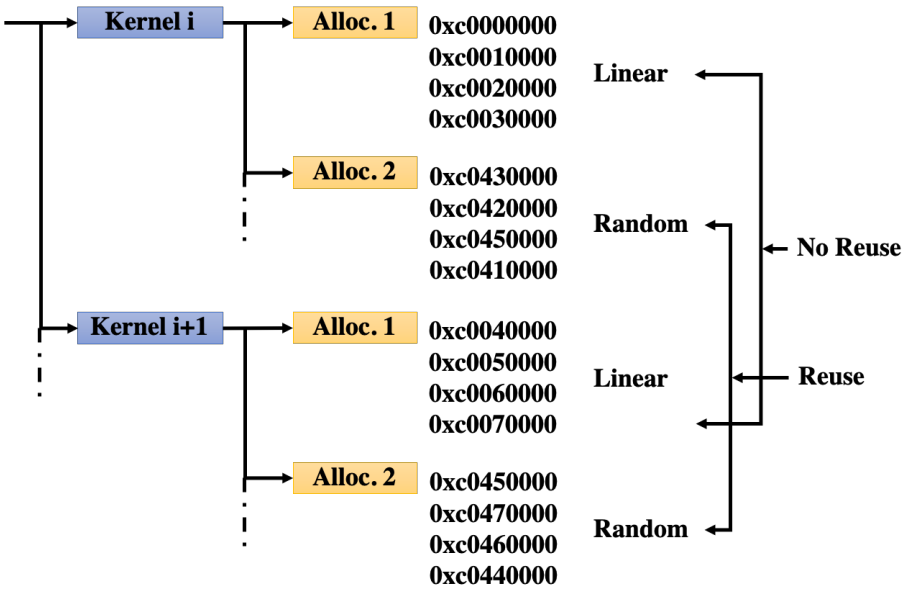



Figure 27: An example of the hierarchical data-structure keeping track of block migration addresses used by detection engine.

① As described in Section 2.2.1, GPU interrupts runtime in the host to relay the far-fault information. ② Based on the group of faults, the software prefetcher determines the $64KB$ basic blocks from the tree structures as migration candidates. ③ These basic blocks are communicated to the I/O root complex to schedule DMA transfers. ④ The pattern detection module of the proposed runtime leverages the information of this basic-block migration information. Throughout computation and data-migration, the runtime keeps a list of the basic block addresses with their corresponding schedule timestamp. ⑤ Runtime is already aware of the base address and allocation size of the managed allocations along with the kernel launch and completion time. ⑥ ***The detection engine is triggered***

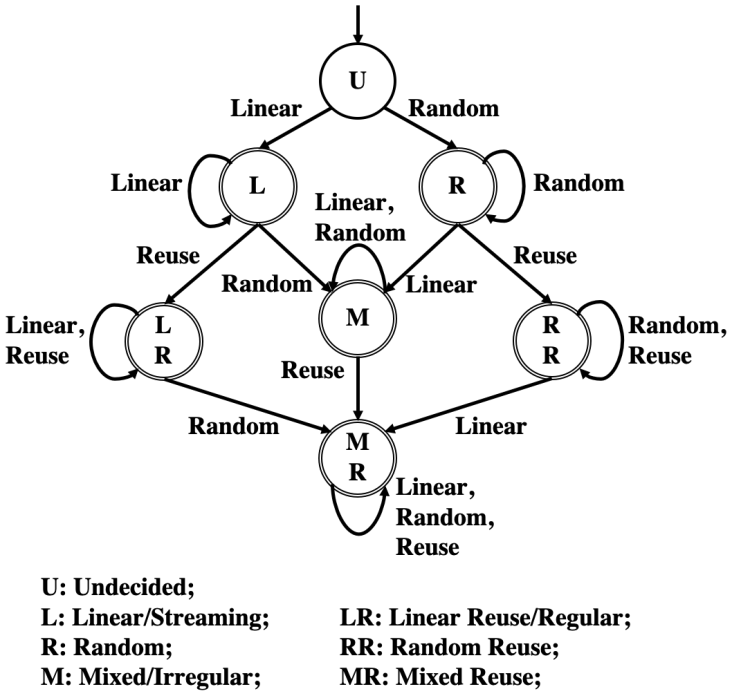



Figure 28: Deterministic Finite Automaton (DFA) for managed allocations demonstrating the transition of migration states.

***at oversubscription before evoking the page eviction routine.*** ⑦ The detection module first scans through the list of basic block transfers and segregates them at kernel boundaries based on their schedule timestamp. Then, within each kernel boundary, migrated blocks are further subdivided into groups of managed allocations based on their virtual address. Figure 27 shows an example of the hierarchical data-structures tracking the basic blocks of migrations grouped by the managed allocation in each kernel boundary. ⑧ Then, the detection module scans through the list of addresses in each managed allocation within the kernel boundary to determine whether they show linearity/randomness of migration. ⑨ Across the kernel boundaries, addresses are compared to determine any re-referencing. ⑩ Based on intra-kernel pattern detection in Step 8 and inter-kernel detection of re-referencing Step 9, the detection module refines the state of managed allocations individually. Figure 28 shows the possible state transition starting from the initial *Undecided* state.

### 6.2.3 Adaptive Memory Management

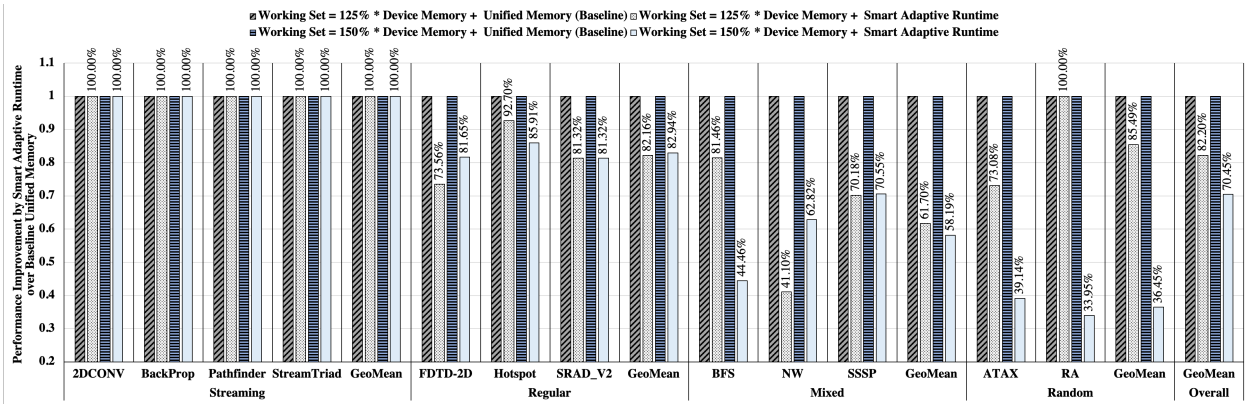
The proposed runtime is the culmination of the lessons learned from the past researches on unified memory oversubscription [32, 31]. Based on the verdict of the pattern-detection module, the policy-engine chooses from the following set of memory management strategies and employs it dynamically. As computation 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 [32] showed the combination of tree-based prefetcher and tree-based pre-eviction can reduce page thrashing for allocations with both regular and irregular migration patterns.

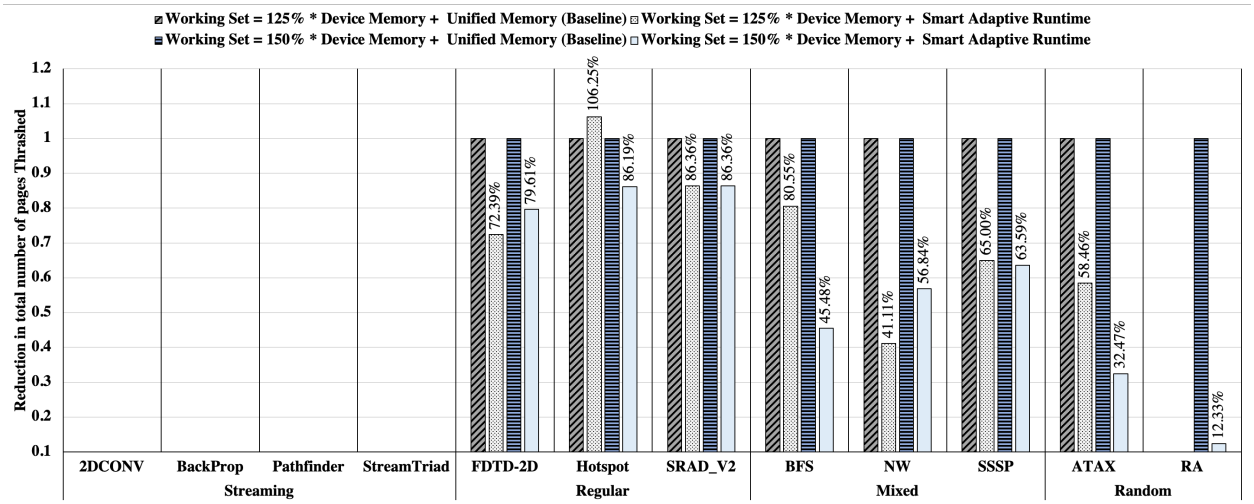
**Streaming/Linear with no data-reuse.** As there is no data reuse and a large array is scanned linearly, the goal for memory management here is to replace memory pages at the highest granularity of $2MB$ huge-pages. This is because the performance overhead of device-memory oversubscription for streaming 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 the device to 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 5, 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 [31]. However, tree-based pre-eviction is employed instead of access counter-based LFU with $2MB$ granularity to reduce page-thrashing.




(a) Performance Speedup




(b) Reduction in Thrashing

Figure 29: Performance of smart adaptive framework compared to unified runtime

## 6.3 Experimental Evaluation

To evaluate the extended framework, the execution time of benchmark kernels running with the proposed smart adaptive framework is compared against the kernels running with the NVIDIA's default unified memory runtime. Figure 29a 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 migration pattern, the performance or smart adaptive runtime is the same as the default unified memory. This is because as described in Section 6.2.3 the memory management strategies are the same for the two. Whereas, for applications with other migration patterns, smart runtime is extremely effective compared to default unified memory. By dynamically determining an adaptive memory management strategy, the smart runtime ensures a considerable reduction in page thrashing as shown in Figure 29b which contributes to the performance speed-up. Note as mentioned earlier, Figure 29b shows no page thrashing for streaming applications due to zero-reuse of data.

## 6.4 Conclusion

The aim of advancing science is to build on existing research by proposing simple yet effective extensions. This chapter is a culmination of the ideas and unification of the memory management techniques presented in the Chapters 4 and 5. This chapter builds on existing runtime and hardware capabilities. Thus, it is not only effective but also has low adaptation costs. Moreover, being application-transparent, the presented smart runtime offers higher programmability with low-performance overhead.

## 7.0 Adaptive Interconnect Provisioning for Multi-tenant Workloads

The past decade has witnessed a steady increase in GPU compute density both in the number of streaming multi-processors (SM) and the number of scalar cores per SM. As a result, there is a growing trend in sharing GPU between concurrent applications to fully saturate available compute-resources and improve value per dollar spent on commodity cloud platforms. However, workload consolidation must guarantee overall system throughput and fairness of execution by eliminating interference between concurrent applications. In the past, researchers have considered partitioning compute resources either spatially or temporally to ensure isolation. They have also investigated provisioning on-chip interconnect bandwidth to remove interference between applications with disparate bandwidth demands. However, with the advent of unified memory, CPU-GPU interconnect becomes a critical resource of consideration as the performance bottleneck is shifted to on-demand page migration.

This chapter goes beyond memory access pattern and characterizes workloads based on their arithmetic intensity and the number of network traffic per unit of unified memory. It demonstrates the limitations with existing interconnect arbiters and traffic schedulers. Based on the key observations, it introduces an adaptive network traffic scheduler that improves system throughput and instruction throughput, while ensuring fairness between participating applications.

### 7.1 Motivation

This section starts with the characterization of general-purpose GPU workloads. Then, it demonstrates the interference between concurrent applications followed by the discussion on the limitations of different interconnect scheduling strategies. To demonstrate the application interference and limitation of scheduling schemes, this section uses two important performance metrics - (i) weighted speedup, and (ii) instruction throughput, which are described in details in Section 7.3.1.

### 7.1.1 Workload Characterization

L2 misses per kilo-instruction (MPKI) is widely used for characterizing memory intensity of applications as well as a proxy for performance [25, 45, 44, 66]. Jog et al [43] showed MPKI alone is not sufficient while categorizing on-chip interconnect and device memory interference for concurrent GPGPU applications. They proposed the bandwidth demand of individual applications as a key performance metric. Several previous works [32, 31, 91, 90] have characterized applications based on their device-memory access pattern. Two broad classes, that emerged from these studies, are - (i) regular, and (ii) irregular. Regular applications have sequential, repetitive access; while irregular applications access memory randomly and sparsely.

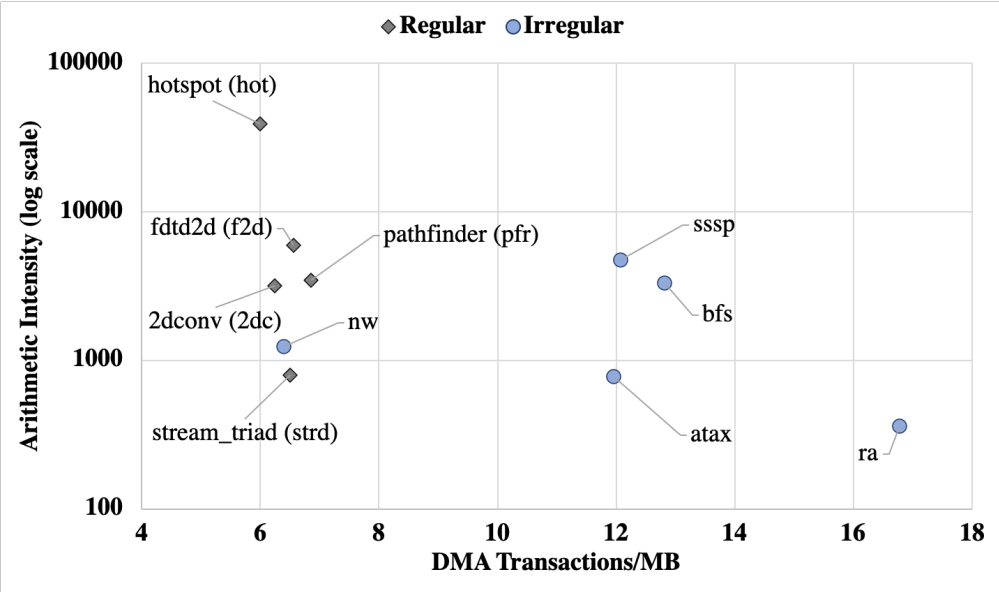



Figure 30: Categorizing workloads based on memory access pattern, arithmetic intensity, and number of DMA transactions per unit memory.

This section shows that while previous works considered regularity(/irregularity) of device-memory access pattern for making page-migration choices to improve CPU-GPU interconnect-bandwidth utilization, such characterization is not sufficient. Figure 30 presents

a set of general-purpose applications, described in Section 3.2.3, and characterizes them based on - (1) number of DMA transactions per unit memory, and (2) arithmetic intensity.

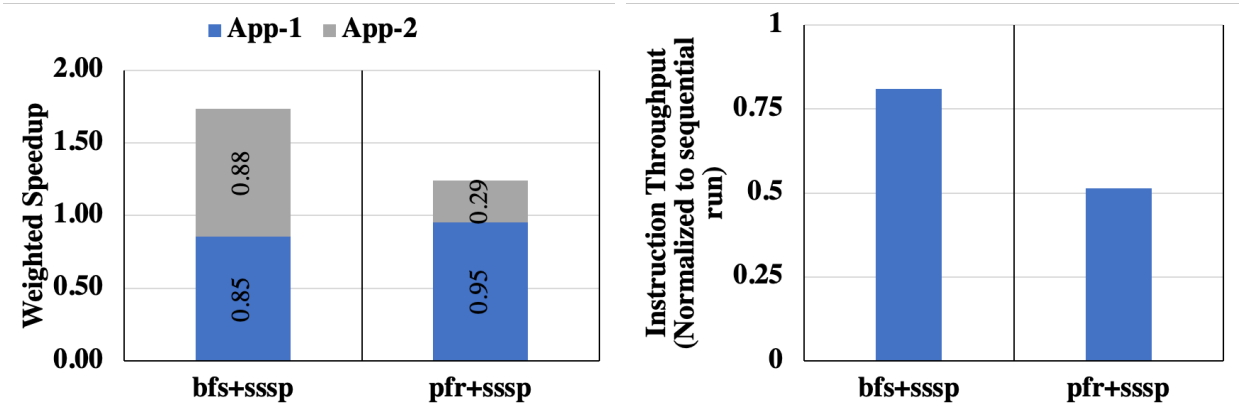
The number of DMA transactions per unit memory allocation is a proxy for the bandwidth demand of an application. It shows the efficacy of the software prefetcher in unified memory runtime. Typically, the software prefetcher works better with the applications with regular memory access pattern than the irregular ones. When the prefetcher under-performs, the runtime creates a lot of DMA requests with smaller memory sizes. As a result, applications with a higher number of DMA transactions per unit memory allocation attain low bandwidth compared to peak interconnect bandwidth. For example, `ra` has higher bandwidth demand or lower bandwidth utilization compared to `2dconv`. Figure 30 presents an interesting observation that while all regular applications belong to a single cluster based on the number of DMA transactions per unit memory allocation, irregular applications create three distinct clusters - (1) `nw`, (2) `atax`, `sssp`, `bfs`, and (3) `ra`.

Arithmetic Intensity (AI) is used in the Roofline model and is expressed as `flops/bytes`. AI expresses how much work is done per unit memory traffic. For applications using a unified memory, AI demonstrates the application's compute-intensity and efficacy of GPU's multi-threading to hide the latency of fault-driven migrations. For example, `hotspot` has higher AI than `stream_triad` as shown in Figure 30. An application with higher AI creates a fewer number of interconnect-transfers in unit time compared to an application with lower AI. Thus, lower AI indicates the higher bandwidth demand of an application.

### 7.1.2 Application Interference

Concurrent applications sharing the same GPU interfere at various levels of the memory hierarchy (e.g. last level cache, device memory) and on-chip interconnect even when the SMs are evenly spatially partitioned among them. However, with unified memory, the performance bottleneck shifts to CPU-GPU interconnect traffic.

Figure 31a shows the performance slowdown of individual applications in a consolidated environment compared to isolated execution with full compute-resources and interconnect bandwidth. `bfs` and `sssp` have negligible slowdown and the total weighted speedup of the




(a) Effect on performance slowdown

(b) Effect of instruction throughput

Figure 31: Effect of application consolidation on system throughput. FR-FCFS is the default scheduling policy.

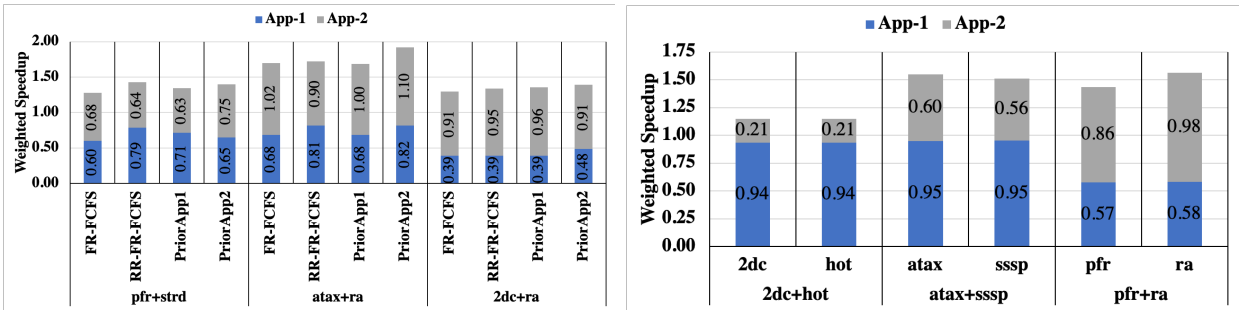
consolidated run is close to the maximum value, 2. `bfs` and `sssp` both belong to the same cluster based on their AI and the number of DMA transactions per unit memory. Whereas, `sssp` experiences a considerable slowdown when executed alongside `pathfinder` as they are diverse in their execution and memory characteristics.

Figure 31b presents the total application throughput of consolidated run normalized by the summation of throughput of applications running in isolation. Like WS, degradation in total IT also shows a strong dependency on the nature of participating applications.

### 7.1.3 Limitations of Existing Scheduling Schemes

First-Ready First-Come-First-Serve (FR-FCFS) [69, 68] is the default scheduling policy of interconnect-arbiters in modern systems. The FCFS nature of the scheduler allows more traffic from the higher memory demanding application as the runtime schedules more network packets in the system queue. As a result, these applications get a higher share of network bandwidth. Whereas RR-FR-FCFS [96] alternates between the network traffics from different applications and thus gives them almost equal service priority. System

administrators also employ static priority for interconnect provisioning. Based on detailed offline-profiling (such as Section 7.1.1), applications with more bandwidth demand are configured to have strict static provisioning priority. Figure 32a shows the performance slowdown of an individual application under the above three scheduling schemes.




(a) Effect of different DMA scheduling

(b) Effect of warp scheduling and in turn order of interrupts (DMA scheduling policy is FR-FCFS)

Figure 32: Different performance slowdowns experienced when different interconnect scheduling schemes are employed.

**Limitations of FR-FCFS.** As discussed above, FR-FCFS allows a higher share of interconnect bandwidth to the more demanding application in the consolidation. For example, this is clear for both `atax+ra` and `2dconv+ra`, as `ra` creates more DMA transactions per unit memory, it hogs the network queue. As a result, both `2dconv` and `atax` suffer significant performance degradation.

**Limitations of RR FR-FCFS.** In case of both `pathfinder+stream_triad` and `atax+ra`, `pathfinder` and `atax` gain performance by 19% and 13% respectively while `stream_triad` and `ra` experiences 4% and 12% slowdown. However, for both of these workloads, prioritizing the latter application over the former improves both of their performance over FR-FCFS.

**Limitations of static application priority.** It can be seen that prioritizing `ra` over `atax` significantly improve the performance of both. However, static provisioning needs in-depth knowledge of application characteristics before consolidating for execution.

**Effect of interrupt ordering and page-fault notification.** As described in Section 2.2.1, the software prefetcher in the runtime schedules host-to-device DMA transactions based on the page-faults notified by GMMU interrupts. The order of page-fault interrupt explicitly depends on the order of warp scheduling and the SM partition on which TBs from a particular application kernel is mapped to. As shown in Figure 32b, prioritizing page-fault notification from one application over the other has no significant impact on speedup for `2dconv+hotspot` and `atax+sssp`. However, for `pathfinder+ra`, prioritizing interrupt notification for `ra` improves overall system throughput.

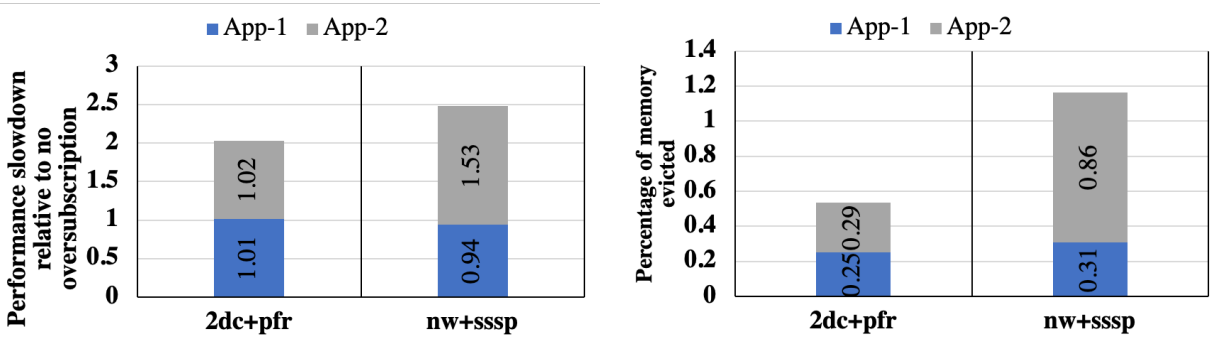
### 7.1.4 Interference under Device Memory Oversubscription

After receiving page-fault notifications from the GPU memory management unit (GMMU), the runtime queries whether there is enough space in device memory for the new page migrations. When the working set exceeds the device memory capacity, the runtime invokes the LRU eviction routine to write-back older $2MB$ huge-pages from the device to the host memory. Under workload consolidation, one application can significantly impact the performance of the other concurrent applications by evicting their memory pages unnecessarily, even without their knowledge.

Figure 33a shows the performance slowdown of the individual application under 110% device-memory oversubscription compared to no oversubscription along with the percentage of allocated memory evicted per application reported in Figure 33b. In the case of `2dconv+pathfinder`, both applications experience similar performance slowdown (1% and 2%) with 25% and 29% of their respective working sets are evicted. However, for `nw+sssp`, two applications experience drastically different performance slowdown which can be attributed to their eviction percentages.

### 7.1.5 Key Observations

This section enumerates the key observations based on the above discussion to build the foundation towards designing an application-aware, adaptive CPU-GPU interconnect




(a) Effect on performance slowdown

(b) Effect of page eviction

Figure 33: Unwanted page eviction and in turn performance slowdown by application consolidation under device-memory oversubscription. App-1 and App-2 are respectively high- and low-priority application chosen by adaptive host-to-device interconnect scheduler.

provisioning scheme.

**Observation 1: Prioritizing applications with a higher number of DMA transactions improves system throughput.** In the workloads, where any participating application suffers significant performance slowdown, prioritizing application with more DMA requests improves overall system throughput. For example, although `atax` and `ra` belongs to the same cluster based on their arithmetic intensity, `ra` creates more DMA requests per unit memory and has higher bandwidth demand. Thus, prioritizing `ra` over `atax` improves system throughput.

**Observation 2: Prioritizing applications with lower AI improves system throughput.** In the workloads, where any participating application suffers significant performance slowdown, prioritizing application with lower AI improves overall system throughput. In the case of `pathfinder+stream_triad`, `stream_triad` has significantly lower AI than `pathfinder` although they create a similar number of DMA requests. Lower AI means `stream_triad` consumes memory faster by doing less computation per unit memory and shows higher bandwidth demand. Thus, prioritizing `stream_triad` over `pathfinder` improves system throughput.

**Observation 3: Order of page-fault notification is as important as the order of DMA requests.** In the workloads, where any participating application suffers significant performance slowdown, prioritizing page-fault notification for application with higher bandwidth demand improves system throughput. `ra` has lowed AI and creates more DMA requests than `pathfinder`. Thus, as shown in Figure 32b, prioritizing interrupt notifications for page-faults in `ra` improves the overall system throughput with the same DMA scheduling policy.

**Observation 4: Preventing page eviction from higher priority application improves system throughput.** In case of device-memory oversubscription, allowing page-migration of one application to interfere with the eviction decision of another application causes serious performance degradation. The degree of performance degradation depends on the nature of participating applications. As prioritizing the high-priority application with higher bandwidth demand improves overall system throughput, the goal is to restrict the eviction of memory pages from high-priority applications giving it the illusion of no memory oversubscription.

## 7.2 Application-aware CPU-GPU Interconnect Provisioning

### 7.2.1 Performance Model

From the discussion in Section 7.1, it can be concluded that maximum attainable bandwidth, $BW_i$ for $i^{th}$ application is a function of $AI_i$ and $T_i$, where $AI_i$ and $T_i$ are respectively arithmetic intensity and the number of DMA transactions per unit memory. More specifically, (1) higher $T_i$ translates to lower $BW_i$ as prefetcher is ineffective resulting fragmented transactions of smaller sizes and (2) higher $AI_i$ translates to lesser number of DMA transactions in unit time resulting higher $BW_i$. This can summarized as Equation 6.

$$BW_i \propto \frac{AI_i}{T_i} \tag{6}$$

Firstly, based on the performance model proposed by Guz et al [37], performance of $i^{th}$

application, $P_i^t \propto BW_i \forall i$ at any given time $t$. $BW_i^{alone}$ and $P_i^{alone}$ are the attained bandwidth and performance of the application when executed in isolation. Let us assume that at time $t+1$, an additional $\varepsilon$ bandwidth is provisioned to the first application by taking it away from the second application. Then, the performance of the two applications at $t+1$ will be $P_1^{t+1} \propto BW_1 + \varepsilon$ and $P_2^{t+1} \propto BW_2 - \varepsilon$ respectively.

In order to improve overall system throughput from time $t$ to $t+1$,

$$\begin{aligned} \frac{P_1^{t+1}}{P_1^{alone}} + \frac{P_2^{t+1}}{P_2^{alone}} &> \frac{P_1^t}{P_1^{alone}} + \frac{P_2^t}{P_2^{alone}} \\ \frac{BW_1 + \varepsilon}{BW_1^{alone}} + \frac{BW_2 - \varepsilon}{BW_2^{alone}} &> \frac{BW_1}{BW_1^{alone}} + \frac{BW_2}{BW_2^{alone}} \end{aligned} \tag{7}$$

Simplifying Equation 7 yields -

$$\begin{aligned} \varepsilon(BW_2^{alone} - BW_1^{alone}) &> 0 \\ \implies BW_2^{alone} &> BW_1^{alone} \text{ , if } \varepsilon > 0 \\ BW_1^{alone} &> BW_2^{alone} \text{ , if } \varepsilon < 0 \end{aligned} \tag{8}$$

Combining Equation 6 and 8, for $\varepsilon > 0$, the following relationships can be concluded -

$$\begin{aligned} T_1^{alone} > T_2^{alone} \text{ , where } AI_1^{alone} \approx AI_2^{alone} \\ AI_2^{alone} > AI_1^{alone} \text{ , where } T_1^{alone} \approx T_2^{alone} \end{aligned} \tag{9}$$

Equation 9 implies that to improve overall system throughput - (1) give more bandwidth to the application with a higher number of DMA requests per unit memory if all the participating applications have similar AI, and (2) give more bandwidth to the application with lower AI if all participating applications generate a similar number of DMA requests. This conclusion is congruent with the observations in Section 7.1.5. Note that this mathematical model can be easily extended to workloads consisting of more than two applications.

### 7.2.2 Mechanism and Implementation Details

Based on the key observations presented in Section 7.1.5 and the analytical model of Section 7.2.1, this chapter proposes an application-aware, adaptive CPU-GPU interconnect scheduling. This scheduler is realized as an extension to the GPU runtime. The scheme has four primary objectives - (i) prioritize traffics from the application with the highest bandwidth demand, (ii) enforce service fairness by avoiding starvation of the applications with lower bandwidth demand, (iii) improve overall interconnect utilization by work-conserving scheduling, and (iv) provide the illusion of no-oversubscription to the high-priority application by avoiding any interference in the eviction decision.

**Determining and Configuring High Priority Application.** Application's bandwidth demand can be quantified by - (i) number of generated DMA requests per unit memory and (ii) AI. DMA transactions are relayed to the root complex and in turn to the interconnect switch by the runtime's software prefetcher logic. The runtime maintains a counter that tracks the number of scheduled transactions per unit size per transaction for the first metric. Measuring AI in the runtime is not trivial as there is no straightforward GPU hardware counter to query committed `FLPOS` like CPU architecture. Note that AI indicates how much time it takes to consume a memory chunk or in other words the frequency of memory requests per unit execution time. The runtime creates a proxy counter for AI that tracks a weighted average transfer size per unit time. The scheduler periodically measures these two counters for each application to determine which application should be prioritized. As discussed previously, if the participating applications have similar AI, then the application with a higher number of generated DMA transactions is prioritized. Similarly, if concurrent applications generate a similar number of DMA requests, then the application with lower AI is prioritized. Based on these two criteria, the scheduler determines a clear high priority application. Then, it sets the TC/VC map field of the VC Resource Control Register. Links in each direction is configured to provision both device-to-host page-fault notification traffic and host-to-device DMA transaction. However, sometimes participating applications can not be strictly classified based on either AI or the number of generated DMA requests. In such cases, failing to determine a clear high priority application, the scheduler falls back to

the RR FR-FCFS by provisioning equal priority for all applications.

**Fair Scheduling.** Application with the highest bandwidth demand hogs the interconnect queues in both FR-FCFS and static priority scheme and thus hurts collocated applications. While a static priority scheme tries to enforce a fixed ratio of the number of serviceable transactions per application, FR-FCFS has no such restriction. Unlike these schemes, the proposed scheduler tries to balance the number of transactions serviced per application by calculating the deviation in performance metrics per application over time. If the performance of the lower priority application deviates by a certain threshold, then its interconnect traffic is prioritized in the future until either the performance is recovered or the higher priority application has a lot of pending serviceable traffics. Thus, the adaptive nature of traffic priority avoids starvation and balances the performance of participating applications.

**Work Conservation.** Sometimes there is no serviceable interconnect traffic from the higher priority application. Instead of enforcing a strict ratio of the number of serviceable transactions per application like a static provisioning scheme, the proposed scheduler allows serviceable traffics from the lower priority application if any. This work-conserving nature of the scheduler improves overall interconnect bandwidth utilization compared to the static scheme.

**Avoid Eviction of High-priority Application Pages.** The goal of the scheduler is to provide the illusion of no memory oversubscription to the high-priority application as if its memory pages are pinned on the device memory. The scheduler prioritizes the eviction of memory pages from low-priority application over the high-priority application. However, it is unfair to prioritize page eviction of the lower-priority application when the high-priority application has already completed its execution. Thus, the runtime engine writes back all memory pages at the last kernel boundary of high-priority application as part of the device-synchronization.

## 7.3 Experimental Evaluation

In total 45 two-application workloads from the 10 unified memory benchmarks part of UMCA described in Section 3.2.3 are evaluated. These workloads are classified in three major categories based on the memory access pattern of the participating applications - (i) `regular+regular`, (ii) `irregular+irregular`, and (iii) `regular+irregular`. Within each category, the workloads are sub-divided based on their respective arithmetic intensity and the number of DMA transactions generated per unit memory. For example, let us consider two workloads within `regular+regular` category - `2dconv+stream_triad` and `2dconv+pathfinder`. All three applications belong to the same cluster based on their number of DMA transactions. However, `2dconv` and `stream_triad` have a large difference in AI. Whereas, `2dconv` and `pathfinder` belongs to the same cluster of AI. Applications in `regular+irregular` category can be further divided based on AI and the number of DMA transactions. While benchmarks like `atax` and `na` have a large difference in both AI and the number of DMA transactions, `atax` and `sssp` belong to the same cluster on both metrics.

### 7.3.1 Evaluation Metrics

To evaluate the concurrent execution of more than one application, one needs to consider the application and overall system throughput along with the fairness of execution between applications.

**Application Throughput.** Instruction Throughput (IT) is expressed as $\sum_{i=1}^{n} IPC_i^{concurrent}$ where, $IPC_i^{concurrent}$ is the number of committed instructions per cycle for the $i^{th}$ application where $n$ applications are executing concurrently. IT measures raw machine throughput.

Overall system throughput can be evaluated using the metric Weighted Speedup (WS), which can be expressed as the summation of slowdown per application compared to isolated run. WS indicates how many jobs are executed per unit time. When there is no interference between the concurrent applications, the WS equals the total number of applications in the composed workload. WS can be expressed as $\sum_{i=1}^{n} SD_i$. $SD_i$ denotes the slowdown of $i^{th}$ application and can be given by $\frac{IPC_i^{concurrent}}{IPC_i^{alone}}$ or $\frac{cyclei^{alone}}{cyclei^{concurrent}}$.

**Fairness.** Kim et al [45] used Harmonic Speedup (HS) to express the notion of fairness between concurrent applications. HS is the reciprocal of Average Normalized Turn-around Time (ANTT) and is expressed as $1/\sum_{i=1}^{n} \frac{1}{SD_i}$.

### 7.3.2 Effect on System Throughput

From the set of total 45 workloads, 17 representatives workloads are chosen for evaluation. This experiment evaluates the performance of adaptive interconnect provisioning in contrast to - (i) default FR-FCFS (baseline), (ii) RR FR-FCFS, and (iii) a static priority scheme. Note that the static scheme requires detailed offline profiling to determine which application should be prioritized. In each workload, the application with the highest bandwidth demand or the lowest attained bandwidth is prioritized. For example, in `ra+stream_triad`, `ra` is given higher priority as it generates more DMA transactions.

Figure 34 shows the weighted speedup of the four considered schemes normalized with respect to FR-FCFS. Note that higher normalized WS indicates the efficacy of the scheme. The average performance result of all 45 workloads using geometric mean (GeoMean) is also reported. Overall the adaptive scheme outperforms FR-FCFS, RR FR-FCFS, and static priority scheme. Particularly, the proposed scheduling scheme gives 8% improvement over FR-FCFS.

Now let us carefully investigate three example workloads (highlighted in Figure 34) from each major category of composition. In `pathfinder+stream_triad`, both of the participating applications belong to the same cluster based on the number of generated DMA transactions. However, `stream_triad` has higher bandwidth demand as it has lower AI than `pathfinder`. The software prefetcher yields higher throughput for both applications because of their regular memory access pattern. Thus, prioritizing `stream_triad` hurts `pathfinder` and in turn overall system throughput. As a result, RR FR-FCFS yields better WS than the static priority scheme. The work-conserving, fair scheduling strategy of the adaptive scheme closely follows the RR FR-FCFS and hence yields closely similar performance.

In `atax+ra`, both applications have irregular memory access patterns, and thus prefetch-

ing is rendered ineffective resulting in more DMA requests per unit memory. Due to the high bandwidth demand of both applications, FR-FCFS and RR FR-FCFS do not work well for this group of applications. Although they have similar AI, `ra` creates significantly more DMA requests in short bursts due to its completely random memory access pattern. The adaptive scheme prefers `ra` over `atax` while provisioning interconnect bandwidth. This results in the adaptive scheme having WS identical to that of the static priority policy.

In `2dconv+ra`, the applications belong to an opposite spectrum based on memory access pattern, and thus `ra` creates 2.5× DMA requests compared to `2dconv`. Because of this diverse nature, a consistent performance improvement can be seen between schemes, with the adaptive scheme yielding 32% speedup over FR-FCFS.

Figure 35 shows the normalized instruction throughput of workloads in each major category along with the overall geometric mean of all 45 workloads. Like WS, the adaptive scheme improves IT by 9% compared to the baseline FR-FCFS.

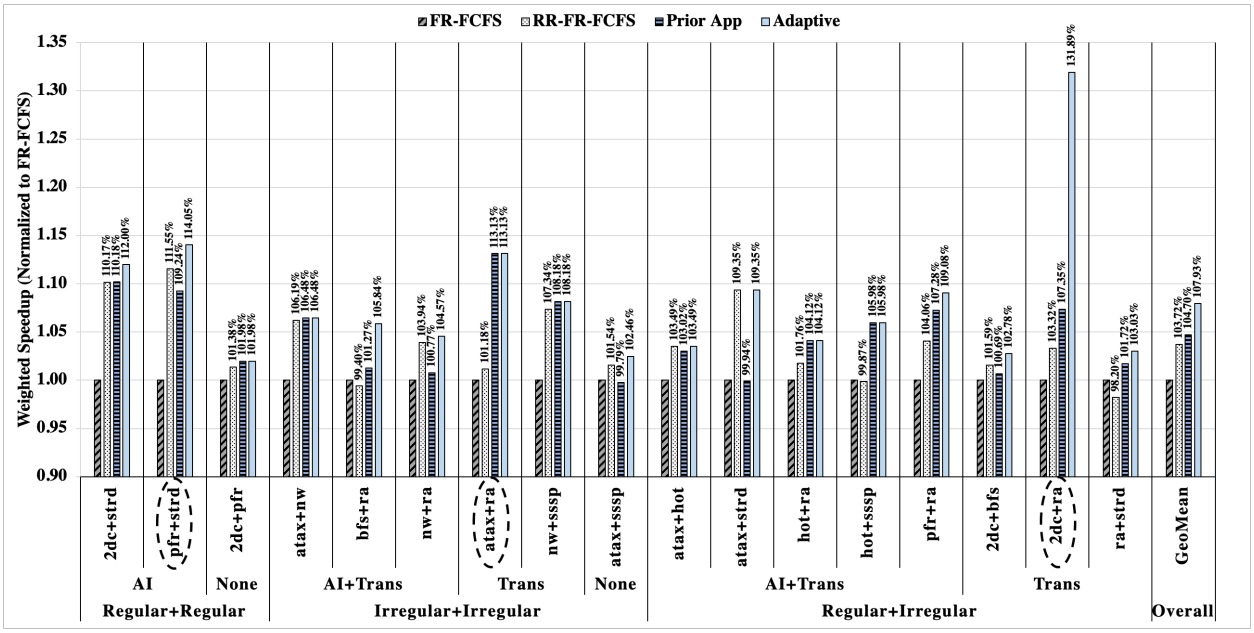



Figure 34: The effect of adaptive CPU-GPU interconnect scheduling on weighted speedup for 17 representative workloads.

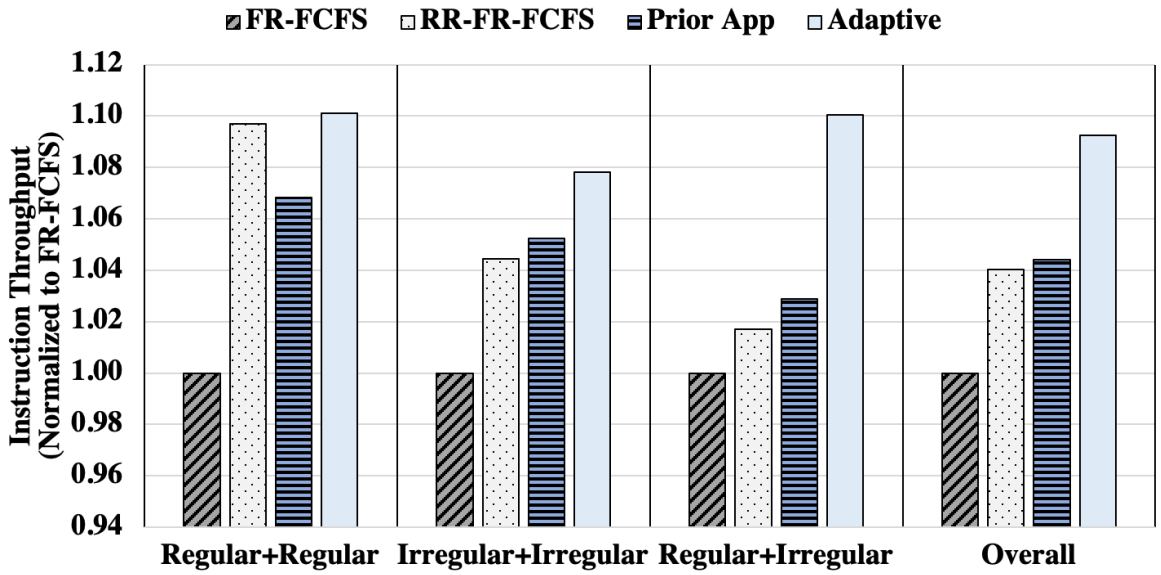



Figure 35: The summary of instruction throughput from adaptive CPU-GPU interconnect scheduling for all 45 workloads.

### 7.3.3 Effect on Fairness

Figure 36 reports the Harmonic Speedup (HS) as a balanced metric for performance as well as fairness. Across all classes of workloads, the adaptive scheme consistently performs better than FR-FCFS with an average (GeoMean) 8% improvement. Careful observation reveals that for the `regular+regular` class, where both applications have high bandwidth utilization, RR FR-FCFS outperforms the static priority scheme and closely follows the adaptive scheme. This proves that the adaptive scheme unlike FR-FCFS and static priority scheme does not starve the application with lower bandwidth demand, rather the work-conserving nature of the scheduler balances the bandwidth demand while ensuring a higher system throughput.

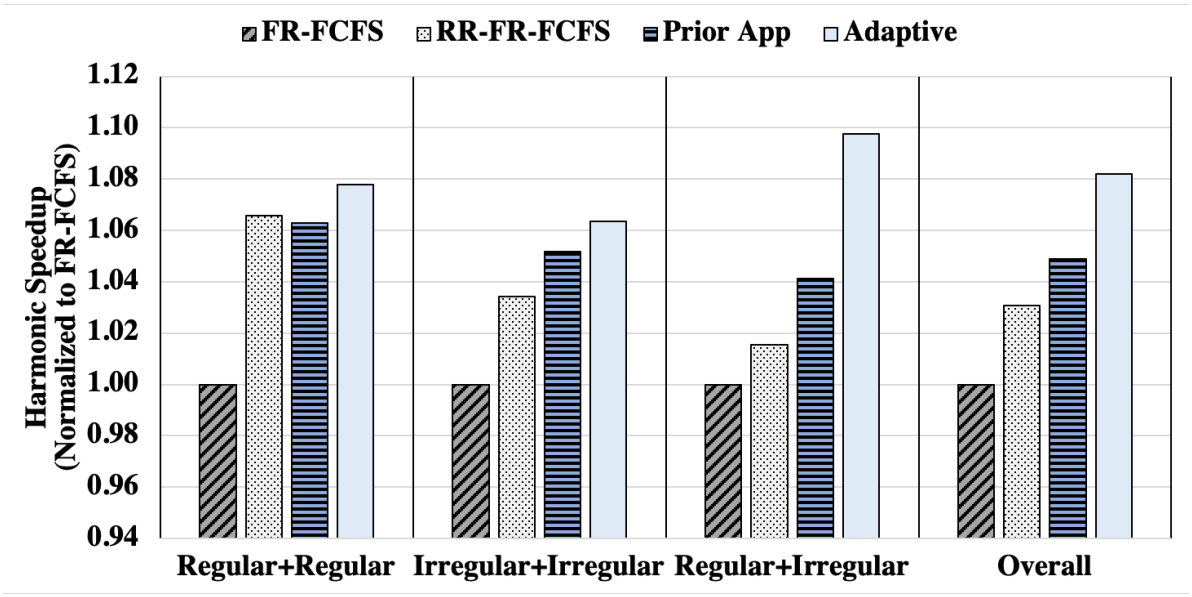


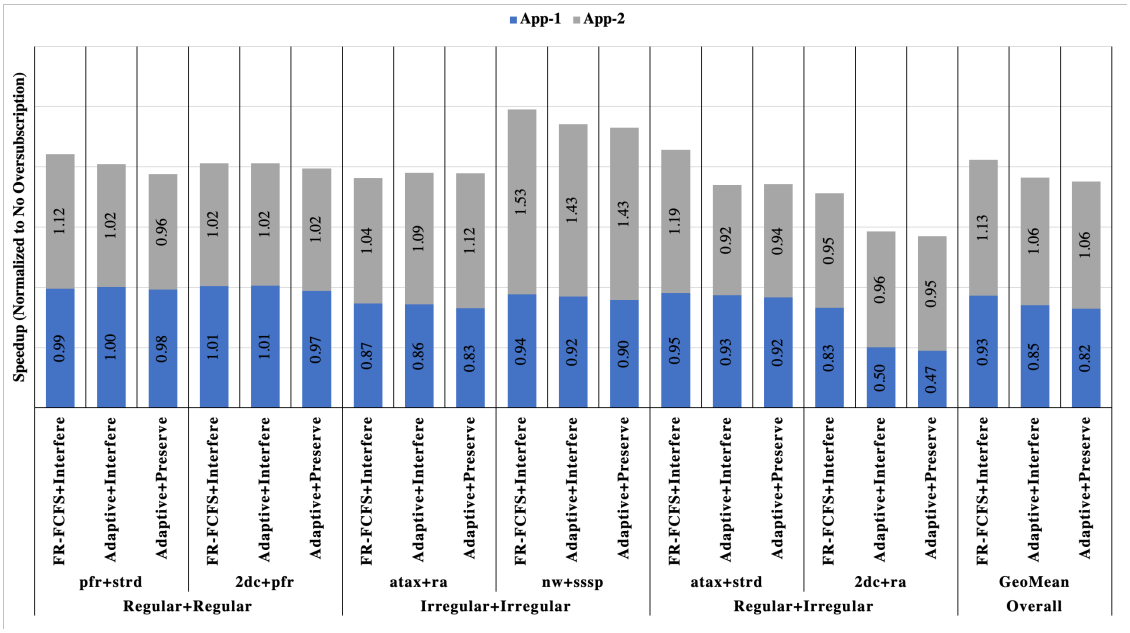

Figure 36: The summary of harmonic speedup from adaptive CPU-GPU interconnect scheduling for all 45 workloads.

### 7.3.4 The Case of Oversubscription

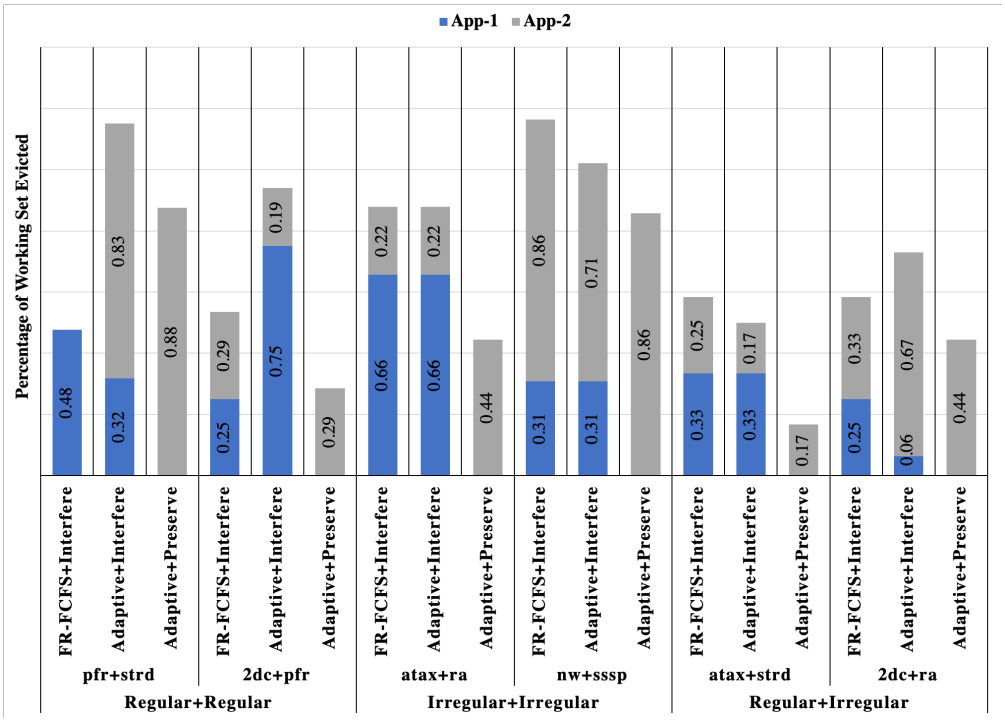
Figure 37 reports the experimental results in the case where the total working set of two concurrent applications is 110% of the device-memory capacity. The labels *App-1* and *App-2* denote higher and lower priority applications dynamically identified by the adaptive host-to-device provisioning scheme respectively. This experiment compares three different interconnect provisioning scheme - (i) **FR-FCFS+Interfere:** the host-to-device DMA traffics and device-to-host interrupt notification traffics are provisioned based on default FR FCFS policy and under oversubscription, device-to-host write-back traffics are not provisioned, (ii) **Adaptive+Interfere:** the host-to-device DMA traffics and device-to-host interrupt notification traffics are provisioned by the proposed adaptive scheduler and the device-to-host write-back traffics are not provisioned, and (iii) **Adaptive+Preserve:** the host-to-device DMA traffics and device-to-host interrupt notification traffics are provisioned by the proposed adaptive scheduler and the device-to-host write-back traffics of lower-priority applica-

tions are prioritized to prevent the eviction of high-priority application's memory pages.

Figure 37a reports the relative slowdown per application compared to their respective performance under no oversubscription or when the working set can fit in the device memory. Figure 37b shows the percentage of allocated memory per application being evicted due to workload eviction under two schemes in consideration. Provisioning host-to-device on-demand migration traffic and device-to-host interrupt notification using adaptive scheduler improve performance of high- and low-priority applications by an average (geometric mean) 8% and 7% respectively resulting in a total 15% gain in overall system throughput compared to the default interconnect provisioning in both directions. Moreover, preserving memory pages of high-priority application from eviction by prioritizing eviction of the low-priority application improves the performance of the high-priority application by an additional 3% without affecting the performance of the low-priority application. Thus, the adaptive provisioning with priority preserving page eviction leads to an average 18% improvement of overall system throughput compared to the baseline. This improvement in performance can be attributed to the reduction in page eviction of high-priority application. Early completion of higher-priority application due to lesser page eviction frees the interconnect bandwidth in both directions for the lower-priority application. Moreover, as the higher priority application finishes, the runtime can now freely evict pages from it causing no further eviction of remaining low priority application at kernel-level device-synchronization. This results in the overall improvement of system throughput.




(a) Effect on Performance Speedup




(b) Effect on percentage of memory being evicted

Figure 37: The effect of priority preserving device-to-host CPU-GPU interconnect provisioning on weighted speedup and page eviction for 6 representative workloads.

## 7.4 Conclusion

State-of-the-art GPU runtime does not have proper support for CPU-GPU interconnect provisioning to ensure system throughput and fairness when a GPU is shared between multiple concurrent applications using unified memory. Firstly, this chapter demonstrates that the current PCIe arbiter is not sufficient to eliminate performance interference between interconnect traffics arising from different applications. Next, it presents an adaptive interconnect traffic scheduling that goes beyond the memory access pattern of participating applications and considers arithmetic intensity and the number of DMA requests to arbitrate priority of CPU-GPU interconnect traffic. Evaluation results show that the proposed scheduler improves overall system throughput and also ensures fairness for a wide variety of workload consolidation.

# 8.0 Concluding Remarks

Unified memory runtime simplifies the memory management of the CPU-GPU heterogeneous memory system and at the same time provides higher programmability. GPUs are classically used to accelerate graphics rendering. However, due to large thread-level parallelism and advancement in the software-managed runtime, GPUs are increasingly being used to accelerate general-purpose applications with large working sets. This exposes new challenges with memory management. This dissertation explores new application-aware heuristics to mitigate performance overhead and improve overall system throughput with minimum modification to the software-managed GPU runtime.

## 8.1 Summary

To address the performance slowdown under device memory oversubscription, the dissertation first introduces a tree-based pre-eviction algorithm. By adopting the semantics of the tree-based prefetcher, it navigates between the two extremes of static eviction granularity of $4KB$ and $2MB$ and thus overcomes the limitations of these schemes. Moreover, by following the spatio-temporal locality of prefetching, it reduces the amount of page thrashing.

Secondly, this dissertation introduces a programmer-agnostic runtime that leverages the hardware access counters to automatically categorize memory allocations based on the access pattern and frequency. The proposed heuristic adaptively navigates between remote zero-copy access to host-pinned memory and first-touch page migration based on the trade-off between low latency remote access and high-bandwidth local access. By dynamically controlling page migration and pinning, the heuristic reduces page thrashing for data-intensive applications with irregular, sparse memory accesses under device-memory over-subscription.

Finally, considering the ever-growing spectrum of general-purpose algorithms in GPU and their diverse memory management needs, this dissertation extends by introducing a pattern detection engine. Based on the underlying CPU-GPU interconnect traffic-access

pattern of the workload, the smart runtime applies the best-suited memory management strategy to reduce page-thrashing.

To extend the memory-management beyond isolated application execution on a GPU, this dissertation delves into the aspect of sharing the GPU by concurrent applications. It proposes a CPU-GPU interconnect scheduler that guarantees execution fairness and improves overall system throughput by provisioning network traffic from the application with higher bandwidth-demand.

## 8.2 Future Direction

Due to the fundamental fact that GPUs are used as slave I/O device to the host processor, the GPU memory management heavily relies on smart memory management techniques in the software-managed runtime. To solve large workloads such as scientific computing, training deep neural networks, application developers are using multi-GPU nodes and even clusters of GPU-enabled nodes connected over a high-performance computer network. Although this dissertation explores memory management for a single CPU-GPU node, the heuristics introduced in this dissertation can be easily adapted to a single-node multi-GPU system with little to no modification. A natural direction for future research is to study the applicability and scalability of these memory management techniques and smart runtime for multi-GPU applications. However, GPU-enabled multi-node systems do not have support for unified memory with the illusion of a contiguous, byte-addressable, virtual address space unlike single-node multi-GPU systems, and GPUs are unfortunately forced to communicate through the driver stack in the host CPU using high-level API calls like MPI. This dissertation calls for attention towards developing a byte-addressable unified virtual address space spanning multiple nodes by leveraging the host-bypass RDMA network. Both migrating pages across physical nodes and directly accessing a memory module in a far-node over the network add to the heterogeneity of memory access. Ensuring data-locality and minimizing network traffic requires careful investigation for multi-node installations further necessitating the development of a smart, adaptive runtime.

## Bibliography


[1] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[2] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. Unlocking bandwidth for gpus in cc-numa systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365. IEEE, 2015.

[3] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *ACM SIGPLAN Notices*, pages 607–618. ACM, 2015.

[4] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. Fair share: Allocation of gpu resources for both performance and fairness. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 440–447. IEEE, 2014.

[5] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. Qos-aware dynamic resource allocation for spatial-multitasking gpus. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731. IEEE, 2014.

[6] Amazon. Amazon EC2 Instance Types. `https://aws.amazon.com/ec2/instance-types/`, 2019.

[7] AMD. AMD APP SDK OpenCL Optimization Guide. `http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf`, 2015.

[8] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 52–61. IEEE, 2001.

[9] ARM. ARM Mali GPU OpenCL Developer Guide. `http://infocenter.arm.com/help/topic/com.arm.doc.100614_0303_00_en/arm_mali_gpu_opencl_developer_guide_100614_0303_00_en.pdf`, 2017.

[10] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: a gpu memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150, 2017.

[11] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330. IEEE, 2010.

[12] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[13] Rishiraj A Bheda, Jason A Poovey, Jesse G Beu, and Thomas M Conte. Energy efficient phase change memory based main memory for future high performance systems. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[15] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM, 2010.

[16] William Bolosky, Robert Fitzgerald, and Michael Scott. Simple but effective techniques for numa memory management. *ACM SIGOPS Operating Systems Review*, 23(5):19–31, 1989.

[17] Timothy Brecht. On the importance of parallel application placement in numa multiprocessors. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 1–18, 1993.

[18] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[19] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. Managing dram latency divergence in irregular gpgpu applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 128–139. IEEE, 2014.

[20] Niladrish Chatterjee, Manjunath Shevgoor, Rajeev Balasubramonian, Al Davis, Zhen Fang, Ramesh Illikkal, and Ravi Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. IEEE, 2012.

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

[22] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.

[23] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. Memory latency reduction via thread throttling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 53–64. IEEE, 2010.

[24] Jonathan Corbet. Autonuma: the other approach to numa scheduling. *LWN. net*, 2012.

[25] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. Aérgia: exploiting packet latency slack in on-chip networks. *ACM SIGARCH computer architecture news*, 38(3):106–116, 2010.

[26] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGARCH Computer Architecture News*, 41(1):381–394, 2013.

[27] Debashis Ganguly. GPGPU-Sim UVM Smart. `https://github.com/DebashisGanguly/gpgpu-sim_UVMSmart.git`, 2019.

[28] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[29] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326. ACM, 2009.

[30] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 7–17. IEEE, 2009.

[31] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Ram Melhem. Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461. IEEE, 2020.

[32] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami 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*, pages 224–235, 2019.

[33] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia De Gonzalo, Thomas B Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 43–54. IEEE, 2017.

[34] Edward H Gornish, Elana D Granston, and Alexander V Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 128–142. ACM, 2014.

[35] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. Ieee, 2012.

[36] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent {GPGPU} kernels. In *Presented as part of the 4th {USENIX} Workshop on Hot Topics in Parallelism*, 2012.

[37] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters*, 8(1):25–28, 2009.

[38] IBM. IBM Power System AC922: Technical Overview and Introduction. `http://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf`. Accessed Apr 04, 2019.

[39] Ilya Granovsky, Elchanan Perlin - IBM. Integrating pci express ip in a soc. `https://www.design-reuse.com/articles/15545/integrating-pci-express-ip-in-a-soc.html`, 2018.

[40] Ravishankar Iyer, Hujun Wang, and Laxmi Narayan Bhuyan. Design and analysis of static memory management policies for cc-numa multiprocessors. *Journal of systems architecture*, 48(1-3):59–80, 2002.

[41] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *DAC Design Automation Conference 2012*, pages 850–855. IEEE, 2012.

[42] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of workshop on general purpose processing using GPUs*, pages 1–8, 2014.

[43] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 223–234, 2015.

[44] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[45] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In

*2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76. IEEE, 2010.

[46] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 28(3):54–66, 2008.

[47] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[48] Nagesh B Lakshminarayana, Jaekyu Lee, Hyesoon Kim, and Jinwoo Shin. Dram scheduling policy for gpgpu architectures based on a potential function. *IEEE Computer Architecture Letters*, 11(2):33–36, 2011.

[49] RP LaRowe, Carla Schlatter Ellis, and Mark A Holliday. Evaluation of numa memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems*, pages 686–701, 1992.

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

[51] Chen Li, Yifan Sun, Lingling Jin, Lingjie Xu, Zheng Cao, Pengfei Fan, David Kaeli, Sheng Ma, Yang Guo, and Jun Yang. Priority-based pcie scheduling for multi-tenant multi-gpu systems. *IEEE Computer Architecture Letters*, 18(2):157–160, 2019.

[52] Wei-Fen Lin, Steven K Reinhardt, and Doug Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50(11):1202–1218, 2001.

[53] Jeffrey C Mogul, Eduardo Argollo, Mehul A Shah, and Paolo Faraboschi. Operating system support for nvm+ dram hybrid main memory. In *HotOS*, volume 9, pages 4–14, 2009.

[54] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of parallel and Distributed Computing*, 12(2):87–106, 1991.

[55] NVIDIA. CUDA Runtime API - v10.0.130. `https://docs.nvidia.com/cuda/cuda-runtime-api/`. Accessed Apr 04, 2019.

[56] NVIDIA. NVIDIA Pascal Architecture. `https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/`. Accessed Apr 04, 2019.

[57] NVIDIA. Multi-process service - volta. `https://docs.nvidia.com/deploy/mps/index.html`, 2019.

[58] NVIDIA Corp. NVIDIA GeForce GTX 1080 Ti. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, 2016.

[59] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving gpgpu concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News*, 41(1):407–418, 2013.

[60] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[61] Milan Pavlovic, Nikola Puzovic, and Alex Ramirez. Data placement in hpc architectures with heterogeneous off-chip memory. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 193–200. IEEE, 2013.

[62] Peter Messmer. Unleash legacy mpi codes with kepler's hyper-q. `https://blogs.nvidia.com/blog/2012/08/23/unleash-legacy-mpi-codes-with-keplers-hyper-q/`, 2012.

[63] Sujay Phadke and Satish Narayanasamy. Mlp aware heterogeneous memory system. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[64] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGPLAN Notices*, pages 743–758, 2014.

[65] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[66] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[67] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[68] Scott Rixner. Memory controller optimizations for web servers. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 355–366. IEEE, 2004.

[69] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2):128–138, 2000.

[70] Nikolay Sakharnykh. 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 Apr 04, 2019.

[71] Nikolay Sakharnykh. 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 Apr 04, 2019.

[72] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74. IEEE, 2007.

[73] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[74] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. Mgpusim: enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 197–209, 2019.

[75] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *ACM SIGOPS Operating Systems Review*, pages 47–58. ACM, 2007.

[76] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. *ACM SIGARCH Computer Architecture News*, 42(3):193–204, 2014.

[77] TOP500.org. Top500 November 2019. `https://www.top500.org/lists/2019/11/`, 2019.

[78] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.

[79] University of Tennesse. HPC Challenge Benchmark. `https://icl.utk.edu/hpcc/`, 2012.

[80] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *ACM Sigplan Notices*, pages 279–289. ACM, 1996.

[81] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S Vetter. Exploring hybrid memory for gpu energy efficiency through software-hardware co-design. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 93–102. IEEE Press, 2013.

[82] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[83] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. Efficient and fair multi-programming in gpus via effective bandwidth management. In

*2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 247–258. IEEE, 2018.

[84] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel: Fine-grained sharing of gpus. *IEEE Computer Architecture Letters*, 15(2):113–116, 2015.

[85] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016.

[86] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 269–281, 2017.

[87] Kenneth M Wilson and Bob B Aglietti. Dynamic page placement to improve locality in cc-numa multiprocessors for tpc-c. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 33–33. ACM, 2001.

[88] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130, 2015.

[89] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242. IEEE, 2016.

[90] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, and Zhiying Wang. Coordinated page prefetch and eviction for memory oversubscription management in gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 472–482. IEEE, 2020.

[91] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. Hpe: Hierarchical page eviction policy for unified memory in gpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[92] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. A quantitative evaluation of unified memory in gpus. *The Journal of Supercomputing*, pages 1–28, 2019.

[93] Jishen Zhao, Guangyu Sun, Gabriel H Loh, and Yuan Xie. Optimizing gpu energy efficiency with 3d die-stacking graphics memory and reconfigurable memory interface. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):24, 2013.

[94] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.

[95] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, pages 129–142. ACM, 2010.

[96] William K Zuravleff and Timothy Robinson. Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order, May 13 1997. US Patent 5,630,096.