AN ABSTRACT OF THE DISSERTATION OF

Yongbin Gu for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on December 2, 2020.

Title: Architecture Optimizations for Memory Systems of Throughput Processors

Abstract approved: _____

Lizhong Chen

Throughput-oriented processors, such as graphics processing units (GPUs), have been increasingly used to accelerate general purpose computing, including machine learning models that are being utilized in numerous disciplines. Thousands of concurrently running threads in a GPU demand a highly efficient memory subsystem for data supply in GPUs. In this dissertation, we have studied the memory architecture of the traditional GPUs and revealed that the traditional memory architecture, initially designed for graphics processing, is less efficient in handling general purpose computing tasks. We propose several memory architecture optimizations for two primary objectives: (1) optimize current memory architecture for more efficient handling of general purpose computing tasks; (2) improve the overall performance of GPUs.

This dissertation has four major parts: (1) The first part deals with the L2 cache inefficiency. A key factor that affects the memory subsystem is the order of mem-

ory accesses. While reordering memory accesses at L2 cache has large potential benefits to both cache and DRAM, little work has been conducted to exploit this. In this work, we investigate the largely unexplored opportunity of L2 cache access reordering. We propose *Cache Access Reordering Tree (CART)*, a novel architecture that can improve memory subsystem efficiency by actively reordering memory accesses at L2 cache to be cache-friendly and DRAM-friendly. (2) The second part deals with miss handling architecture (MHA) in GPUs. Conventional MHA is static in sense that it provides a fixed number of MSHR entries to track primary misses, and a fixed number of slots within each entry to track secondary misses. This leads to severe entry or slot under-utilization and poor match to practical workloads, as the number of memory requests to different cache lines can vary significantly. We propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that dynamically forms MSHR entries from a pool of available slots. This approach can self-adapt to primary-miss-predominant applications by forming more entries with fewer slots, and self-adapt to secondary-miss-predominant applications by having fewer entries but more slots per entry. (3) The third part aims to improve the performance of Unified Virtual Memory (UVM), which is recently introduced into GPUs. We propose *CAPTURE (Capacity-Aware Prefetch with True Usage Reflected Eviction)*, a novel microarchitecture scheme that implements coordinated prefetch-eviction for GPU UVM management. CAPTURE utilizes GPU memory status and memory access history to dynamically adjust the prefetching and "capture" accurate remaining page reusing opportunities for improved eviction. (4) In the fourth part, we propose a comprehensive UVM benchmark suite named

UVMBench to facilitate future research on the UVM research.

Architecture Optimizations for Memory Systems of Throughput Processors

by

Yongbin Gu

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 2, 2020
Commencement June 2021

Doctor of Philosophy dissertation of Yongbin Gu presented on December 2, 2020.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Yongbin Gu, Author

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my advisor, Professor Lizhong Chen, for giving me the opportunity to become a PhD student, for his excellent mentorship and for his encouragement during my studies at Oregon State University. Without his support, it would not be possible to complete this dissertation.

I also would like to thank Professor Bella Bose, Professor Ben Lee, Professor Thinh P. Nguyen and Professor Joseph Louis for their acceptance of being my committee members and valuable comments on my dissertation.

I am very grateful to know many friends at Oregon State University, especially my lab-mates, Yunfan Li, Arash Azizi, Fawaz Alazemi, who helped me from the first day I joined the group.

I also want to express my gratitude to Dr. Pengcheng Li, Dr. Tao Zhang and Dr. Haixin Liu who are the mentors during my internship.

Last but not least, I would like to express my sincere appreciation to my family for their continuous support and encouragement during my study.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

Chapter 1: Introduction

1.1 Graphics Processing Units

Graphics Processing Units (GPUs) are first designed for accelerating image display on monitors. They are widely used in personal computers, mobile devices and game consoles. As the image processing mainly involves in matrix operation, which can be paralleled, the GPUs are implemented with a highly parallel architecture. With the help of the parallel programming APIs (e.g. CUDA[25], OpenCL[38]), their highly parallel structure makes them more efficient than CPUs on general purpose tasks, such as machine learning acceleration and molecular simulation.

Figure 1.1 depicts the general architecture of the modern GPU (Pascal). A GPU have several streaming multiprocessors (SMs), each of which also has multiple cores. The cores in the SMs are composed of ALUs, thread schedulers, load/store units, scratchpads, register files and caches, and so on. As GPUs are first designed for streaming computing (image processing), which usually has little data reuse, therefore, GPUs usually have much smaller cache capacity (e.g. 48KB L1 and 2.75MB L2 for GTX 1080 Ti) than the CPUs. A GPU has its own device memory of a few GBs. And it is connected to the CPU through a PCIe bus. The codes running on the GPUs are called kernels. The kernel is executed on the GPU in groups of 32 threads, called a warp [86].

Figure 1.1: GPU Diagram (Pascal Architecture).

1.2 GPU Memory Access Orders

With massive parallel computing ability, graphics processing units (GPUs) are being increasingly used to accelerate numerous scientific, economic and general purpose computing applications. GPUs employ single instruction, multiple thread (SIMT) architecture, which allows thousands of threads running simultaneously (e.g. up to 3584 threads in NVidia GTX1080 Ti). These concurrent threads

generate a large number of memory requests that put high pressure on the memory subsystem (e.g., cache, on-chip network, DRAM) [49]. If not designed with care, the memory subsystem can easily become a serious factor that prevents GPUs from achieving peak performance. With the current technology and application trends, the issue of memory subsystem will likely worsen in the near future. On the technology side, the development of memory technology have been lagging behind processing, e.g., from NVidia GTX480 to GTX1080 Ti, the core count increases by more than 7.4X, but the DRAM bandwidth increases only by about 1.7X. On the application side, irregular memory access patterns have been exhibited in more and more GPU workloads (such as trees, priority queues, key-value storage [19, 42]), which often have poor cache locality and greatly exacerbate the memory stress. Thus, it is imperative to explore new opportunities in the memory subsystem, particularly at the architecture level, to bridge the gap between technology and application demands.

A key factor that determines the efficacy of memory subsystem at all levels of the memory hierarchy is the order of memory accesses. The order affects not only the hit/miss of the current level, but also determines which accesses are exposed to the next level. While prior research has investigated the access reordering benefits in L1 cache and in DRAM (More details in Related Work), the reordering opportunity at L2 cache has largely been unexplored. Nevertheless, the access order to L2 can have a large impact on both L2 cache and DRAM. On the one hand, the access order can be utilized to extract potential data locality to increase cache hit, as well as to reduce avoidable head-of-line blocking in the request buffer

of L2 cache. On the other hand, the access order also determines the request order to DRAM. A benign request sequence to DRAM offered by L2 can greatly facilitate memory controllers to improve row-buffer hit and bank-level parallelism (BLP), both of which are critical to DRAM performance. Substantial research is needed on how memory accesses can be reordered to achieve a cache-friendly and DRAM-friendly order.

1.3 GPU Memory Miss Handling Architecture

Many-core processors have an increasing demand for higher memory level parallelism (MLP) to achieve better performance [43]. Consequently, a large number of outstanding memory requests need to be tracked simultaneously in the memory subsystem by the miss handling architecture (MHA). This demand becomes more pressing in GPUs, as the single instruction multiple threads (SIMT) model can easily execute hundreds to thousands of threads concurrently, resulting in numerous memory requests pending in the memory hierarchy. Thus, it is imperative to design miss handling architectures that can process and track cache misses at a matching rate.

MHA has been evolving continuously in the past years, with most of today's GPUs having MHA based on *Miss Handling Status Registers (MSHRs)*. When a requested data is not found in the cache and sent to the next memory level, the associated MSHR tracks the cache miss by temporally storing the requester ID, cache block address, requested data tag, and other related information until the

data is returned from the lower level. A typical MHA may have dozens of MSHR entries (e.g., 32 or 64) and each entry may in turn have multiple slots (e.g., 4 or 8). An entry is allocated to the primary miss to a cache line, and the slots within the entry are allocated to the secondary misses to the same cache line while the primary miss is pending. The MHA is critical to memory level parallelism, as no new memory requests can be processed if there is no free entry or slot available in the MHA.

While the above architecture works well to a certain degree, it may no longer be sufficient in handling the increasing diverse miss behaviors in GPU workloads. The main issue with the conventional array-based MSHRs is that the entire structure is static, in the sense that every entry has the same number of slots and this number is fixed after manufacturing. However, it is unlikely that every cache line has the same number of misses. While some entries are in high demand for slots, other entries may have multiple slots being unused. To understand the workload demand in practice better, we evaluated a number of applications from three widely used GPU benchmark suites. Results show that the cache misses in most benchmarks are predominant by either primary misses or secondary misses. This highlights that the entry/slot utilization in conventional MSHRs would be poor when running the common workloads, and that the structure would not perform well for all the applications due to the diverse miss behaviors. A direct and naive way to address this issue is to add more entries and slots. This method not only incurs substantial overhead (e.g., 22.3% overhead in terms of L2 cache area, as shown later), but also has limited effectiveness as certain applications may demand over 30 secondary

misses to the same cache line (thus requiring 30 slots per entry) but only need 2 to 3 entries. It is simply impractical to increase MSHRs from the typical 4-8 slots per entry to that size. To address this important problem, innovative solutions are needed to utilize the MSHR resources smartly.

1.4 Unified Virtual Memory in GPUs

The superior computing capability and improved programmability have increased the popularity of GPUs among high performance applications [46, 39]. However, recent AI algorithms and HPC applications [28, 6, 67, 101] on GPUs have exhibited an ever-increasing demand for memory capacity (e.g., even the advanced Titan X GPU may not be able to run the BERT_{Large} model when sequences become a bit longer [28]). Consequently, the limited GPU memory size [83, 84] and the traditional "*copy-then-execute*" programming model [108] have become major performance bottlenecks for emerging applications. To address this issue, both Nvidia and AMD [33] have integrated the Unified Virtual Memory (UVM) support in their GPUs released since 2017, which enables automatic on-demand page migrations, and hence significantly saves GPU programming efforts and mitigates the physical memory capacity limitation [7, 8, 12, 77, 95].

With the introduction of a new type of page faults in UVM, namely *page far-fault* [108] (i.e., data is not present in the GPU memory and need to fetch from CPU), the hardware can fully take charge of page fault handling if the required data is not present in the device memory. This improvement brings a big relief

for programmers as they no longer need to pay attention to the data presence in the GPU memory. Nevertheless, previous studies [108, 34, 33] show that on-demand page migration (i.e., only migrating required pages that are faulted) can incur severe performance degradation due to frequent, long-time GPU thread stalls caused by enormous page faults. To tackle this issue, page prefetching is adopted as a promising way [108, 33] to reduce page fault occurrences, as most of prefetched pages are accessed by the GPU sooner or later. However, prefetching is a double-edged sword: prefetching an improper numbers of pages at the wrong time may incur a high occupancy of GPU memory and also waste PCIe bandwidth between the CPU and GPU.

1.5 GPU simulators

In the past decades, several open-source GPU simulators has been released, and serves for different purposes. Table 1.1 summarizes current available open-source GPU simulators. GPGPU-Sim[10] is a simulator for Nvidia GPUs. It was developed based on the Fermi architecture, and is capable of executing Nvidia virtual ISA. Multi2-Sim[35] is a versatile simulator which can simulate virtual ISAs from both Nvidia and AMD GPUs. Gem5-APU[13] was developed based on the Gem5 simulator, which is augmented by the AMD APU performance model. It can only simulate AMD virtual and machine ISAs. MGPU-Sim[92] is a parallel GPU simulator. The outstanding feature of this simulator is that it can conduct multi-threaded simulation. Accel-Sim[51] is the most recent update version of the GPGPU-Sim.

Table 1.1: Open-source GPU simulators

	GPGPU-Sim 3.x [10]	Multi2-Sim [35]	Gem5-APU [13]	MGPU-Sim [92]	Accel-Sim [51]
Validated GPU Model	Fermi	Kepler	AMD	AMD	Kepler, Pascal, Volta, Turing
Validate Workloads	14	24	10	7	80
Multi-threaded Simulation	✗	✗	✗	✓	✗
Report Accuracy	35%	19%	42%	5.5%	15%

It extends the performance model of the GPGPU-Sim with recent released NVidia GPU architectures. This simulator can also support trace level simulation. In the following chapter, we mainly use GPGPU-sim and our in-house revised version to validate our proposed schemes.

1.6 Benchmarks for GPU Memory Architecture Research

GPUs have been gaining great attention in accelerating traditional and emerging workloads, such as machine learning, bioinformatics, electrodynamics, etc. due to GPU's massively parallel computing capability. However, there are two major issues in the mainstream GPU programming model that severely limit further utilization. First, the physical memory separation between a GPU and a CPU requires explicit memory management in conventional GPU programming model. Programmers have to explicitly copy data between CPU and GPU memories to the location where the data is used (i.e. *copy-then-execute*). Second, the conventional GPU programming model does not allow a kernel to be executed if it needs more memory that what the GPU memory can provide (i.e., *memory oversubscription*). This has greatly limited the use of GPUs in large data-intensive machine learning applications [28, 101] nowadays. Recently, GPU vendors have proposed and started to employ a new approach, *Unified Virtual Memory (UVM)*, in the newly released products[87, 2]. UVM allows GPUs and CPUs to share the same virtual memory space, and offloads memory management to the GPU driver and hardware, thus eliminating explicit copy-then-execute by the programmers.

The GPU driver and underlying hardware automatically migrate the needed data to destinations. Moreover, UVM enables GPU kernel execution while memory is oversubscribed by automatically evicting data that is no longer needed in the GPU memory to the CPU side. This is extremely important and helpful in facilitating large workloads (especially deep learning models) and GPU virtualization [64, 40] with limited memory sizes.

However, the advantages of UVM may come at a price. Analogous to virtual machines that offer great flexibility over physical machines but sacrifice performance in some degree [107], UVM also incurs performance overhead. In order to implement automatic data migration between a CPU and a GPU, the GPU driver and the GPU Memory Management Unit (MMU) have to track data access information and determine the granularity of data migration over the PCIe link [33]. This may reduce performance. For example, UVM needs special page table walk and page fault handling that introduce extra latency for memory accesses in GPUs. In addition, the fluctuated page migration granularity may also under-utilize PCIe bandwidth.

Due to the large potential benefits of UVM and its associated performance issues, UVM has recently drawn significant attention from the research community. Several optimization techniques have been proposed to mitigate the side effects of UVM [108, 66, 60, 33, 105, 53, 32]. The earliest work is Zheng *et al.* [108], which enables on-demand GPU memory and proposes prefetching techniques to improve UVM performance. As the work predates the release of UVM, the developed on-demand memory APIs are quite different from the version in the current UVM

practice. More recently, Ganguly *et al.* [33], Yu *et al.* [105] and Li *et al.* [60] study prefetching and/or eviction techniques for UVM in more detail. However, their evaluation includes only benchmarks with limited number of access patterns, which makes it difficult to assess the effectiveness of their schemes on a broader range of benchmarks with diverse memory access patterns. In fact, comprehensive benchmarks (or the lack thereof) have become a common issue in these and other prior works on GPU UVM. Most of them have used their own modified versions of existing benchmark suites (e.g., Rodinia [22, 23], Parboil [91], Polybench [78]) or several in-house workloads. Our further inspection of these benchmarks shows that they lack unified implementation and no paper so far has provided a thorough analysis of the memory behaviors of these benchmarks. This can be a serious limitation for researchers and developers who aim to propose new optimizations for UVM and who would like to make comparison with existing research works.

Chapter 2: CART: Cache Access Reordering Tree for Efficient Cache and Memory Accesses in GPUs

2.1 Basic Idea

In this work, we explore the opportunity of reordering memory accesses at L2 cache. We conduct an in-depth analysis on when and why access reordering at L2 can be beneficial to both cache and memory. The challenge, however, is to design a well-rounded reordering architecture that addresses data locality, row-buffer hit, bank-level parallelism and low design cost at the same time.

To address this challenge, we propose *Cache Access Reordering Tree (CART)*, a novel yet effective architecture to reorder memory accesses at L2 cache. The main idea is to classify and group memory accesses by passing the accesses through a reordering tree. The reordering tree takes into account data locality in cache lines to increase cache hit, as well as the bank, row and column information of the accesses to increase DRAM efficiency in case of cache misses. We propose a way to use a very small number of leaf queues to mimic the effects of having a large number of queues to reduce hardware cost. A fill policy and a drain policy for memory requests are carefully designed to make full use of the reordering tree. Cycle-accurate simulations based on a wide range of benchmarks show that, the proposed CART is able to improve the average IPC (geometric mean) of memory

intensive benchmarks by 34.2% with only 1.7% area overhead, compared with the conventional design. Furthermore, CART is able to complement other state-of-the-art techniques on GPU caches to achieve higher performance. For example, when combined with MRPB (Memory Request Prioritization Buffer) [47] and RACB (Resource Aware Cache Bypass)[27], the two combinations can achieve a total improvement of average IPC by 38.6% and 41.5%, respectively.

2.2 Background and Motivation

2.2.1 Memory Subsystem in GPUs

Figure 2.1 depicts a typical GPU architecture and where the proposed CART fits. A GPU mainly consists of streaming multiprocessors (SMs), interconnect network, L2 cache, and DRAM. An SM has a number of SIMT cores (e.g. 128 cores per SM in NVidia GTX1080 Ti) to execute multiple threads in parallel. For the memory subsystem, L1 cache(s) exists inside each SM and handles requests from multiple SIMT cores within the SM; whereas L2 handles memory requests that are coming from the SMs through the interconnect network. The logically unified L2 cache is split into several partitions and each partition is associated with a DRAM partition. To track multiple outstanding misses to the DRAM, miss status handling registers (MSHRs) are employed to keep track of the needed information for each DRAM request, such as the requester core ID, cache block address, returned data destination, new data for write-back (in case of writing).

For a primary cache miss that requests a new cache block, one MSHR entry is allocated. For a secondary cache miss (that requests data in the same cache block that has been allocated an MSHR entry and is currently pending), one slot in the MSHR entry is allocated, provided that an empty slot is available in that entry. A typical MSHR may have 32 or 64 entries, with each entry having 4 slots.

2.2.2 Impact of Access Order on L2 Cache

The order of memory accesses to L2 cache plays a significant role in determining memory access latency. The access order not only affects the locality of data which in turn influences cache misses, but also has a large impact on the blocking time of memory accesses in the cache. The latter is due to the FIFO nature of the incoming buffers in L2 cache. In conventional GPUs, memory requests that come out of the interconnection networks are enqueued in the incoming buffer of the corresponding L2 cache partition (Figure 2.1 and Figure 2.2(a). When a request moves to the head of the buffer, L2 checks if the request is a hit in the cache; if not, the request needs to be issued to DRAM by allocating an MSHR entry or slot. However, a reservation fail (RF) may happen when no entry/slot is available in the MSHR or when the miss queue to DRAM is full. As a result, the request has to stay in the incoming buffer and retries later. This blocks other memory requests in the FIFO buffer, even though some of the requests could hit in the L2 cache (no need for MSHR) or use MSHR in other ways (more analysis in Section 3.1). This head-of-line blocking is more pronounced for irregular memory accesses that

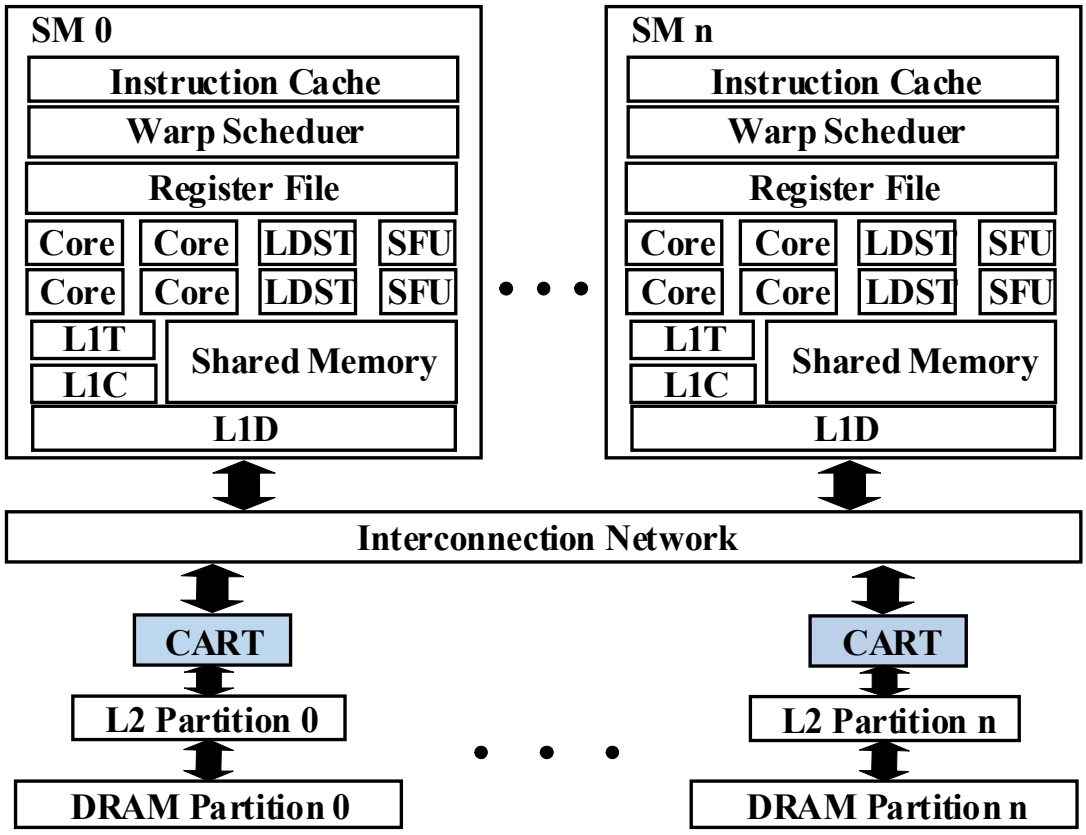


Figure 2.1: A typical GPU architecture (MSHRs in L1 are omitted for clarity). The proposed CART is added before L2.

have burst patterns. One of our goals is to reduce the occurrence of head-of-line blocking without affecting data locality through a better cache-friendly reordering scheme.

2.2.3 Impact of Access Order on DRAM

The order of memory accesses also has a large impact on the efficiency of DRAM because of row-buffer conflicts and bank-level parallelism (BLP). DRAM has a three-level structure, namely banks, rows and columns [20]. For example, a DRAM chip may consist of 16 banks, with each bank having thousands of rows and tens of columns in each row. The size of each column in a row is usually the size of a cache line (e.g., 128 bytes). Therefore, upon a cache miss, the memory address is decoded to locate the correct bank, row and column to fetch an entire cache line (i.e., a column). Modern DRAMs employ a row buffer in each bank that serves as a "cache" function for temporarily storing the contents of one row, so as to accelerate future accesses of columns in the same row. A *row buffer conflict* occurs if the column from a different row is requested, in which the row buffer is flushed and refilled by the contents of the newly requested row. This results in additional access latency. Several memory schedulers (e.g. [72, 85]) try to reduce row buffer conflicts by reordering memory accesses on the DRAM side. However, due to the above-mentioned blocking issue in L2, many memory requests are congested at L2. This leaves a limited number of requests at the front of DRAM for reordering. Hence, it is important to create a benign order of memory accesses early on at the L2 cache. Similarly, as banks in a DRAM chip can work in parallel, it is also beneficial to reorder memory accesses at L2 in a DRAM-friendly way to help distributing memory requests more evenly among different banks to increase parallelism.

2.2.4 Need for More Research on Reordering

To improve the effectiveness of the memory subsystem, several optimization approaches have been proposed, but the opportunity of reordering memory access order at L2 cache has largely been unexplored. One approach is to increase MSHR sizes to reduce reservation fails. However, enlarging MSHR is often prohibitively costly due to its content-addressable memory (CAM) circuitry [94, 44], and not all blocking cases are caused by MSHR size limitation. Additionally, increasing MSHR size does not improve DRAM efficiency as it could not reorder memory requests to lower row buffer conflicts or increase BLP.

In terms of reordering, reordering memory requests at L1D in a cache-friendly order has been proposed to increase cache hits and overall performance [47]. Cache bypassing is used to reduce the penalty of reservation fails [104, 27, 59]. Researchers also propose to reorder through memory schedulers at memory controllers to reduce memory accessing latency and increase DRAM working parallelism [69]. While more related works are discussed in Section 2.7, existing approaches have not explored the reordering at L2 cache, which has large impact on both cache and DRAM as analyzed in the above two subsections. In following sections, we present how a reordering architecture and strategy can be designed at L2 cache to address data locality, head-of-line blocking, row buffer conflict, and bank-level parallelism at the same time.

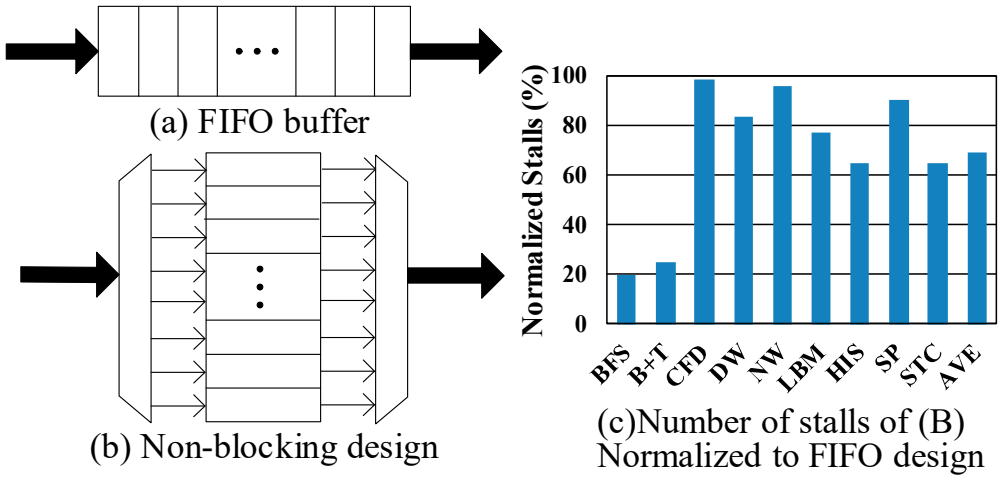


Figure 2.2: Blocking vs. non-blocking request buffers.

2.3 Exploring Access Reordering at L2

2.3.1 Blocking of Memory Requests at L2

The root cause of the blocking issue of memory requests at L2 is the FIFO structure of the incoming buffer. Under such design, if the memory request at the head of the incoming buffer (head request) is stalled, all the subsequent requests are blocked in the buffer. Specifically, there are three cases where removing such blocking may lead to performance benefits: (1) the head memory request is a primary cache miss and is stalled due to the lack of available entry in MSHR; however, a currently blocked subsequent request could have been merged into an existing MSHR entry (i.e., a secondary miss). (2) the head memory request is a secondary cache miss and is stalled due to the lack of available slot in the matching MSHR entry (i.e., needs to be merged with the primary miss); however, empty MSHR entries are

available and could have been allocated to currently blocked subsequent requests. (3) the head memory request is stalled due to reservation fail or DRAM saturation (or any other reasons), but the blocked subsequent requests could have hit in L2 cache and should have proceeded.

2.3.2 A Straightforward Non-blocking Design

To reap the above benefits, we start by considering a simple but non-blocking incoming buffer design that supports any access order. As illustrated in Figure Figure 2.2(b), the incoming buffer is restructured to enable parallel selection of any memory request using a giant multiplexer. When a request encounters a stall, a selection policy (e.g., round-robin) is employed to select the next request that is qualified for draining from the buffer structure. The selected request must not be stalled by the same resource as the previously stalled request. Although being straightforward, this design can significantly reduce the number of stalls at L2 by 68.8% on average, as shown in Figure 2.2(c). Nevertheless, this design has two major drawbacks:

- It only solves the blocking that is local to L2 cache, while neglecting other opportunities in DRAM down the line, such as row buffer hit and bank-level parallelism.
- The arbitration can be quite complex, as the multiplexer and control logic need to scan through all the requests in the buffer to identify a qualified draining candidate.

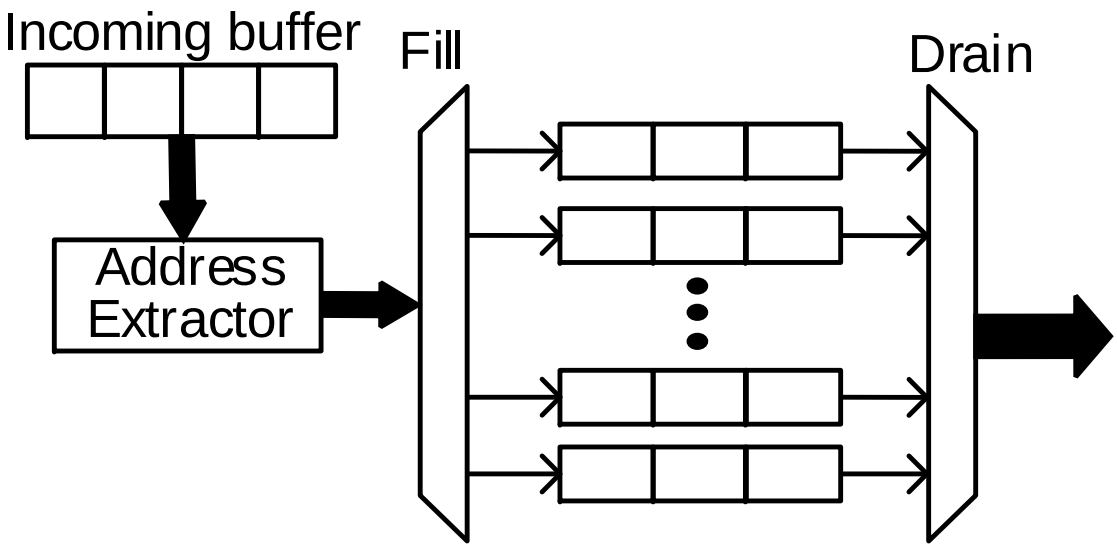


Figure 2.3: Reordering based on bank information.

2.3.3 An Improved Design for Access Reordering

To tackle these problems, we examine an improved design that takes into account bank-level parallelism and arbitration. As shown in Figure 2.3, in this design, the FIFO incoming buffer remains the same, but B FIFO queues are added to classify memory requests that come out of the incoming buffer. A simple address extractor extracts the bank address from a given memory request, and directs the request to one of the FIFO queues by calculating (bank_address mod B).

Note that if B equals the number of banks, memory requests are essentially queued by their bank addresses. However, B can be less than the number of banks, in which case memory requests destined to different banks may share a queue. Finally, for draining, a round-robin policy is used to select a non-empty

queue among the B queues in each cycle.

Compared with the parallel design in Figure 2.2(b), DRAM bank-level parallelism is improved because every time a memory request is selected to drain, its bank address is guaranteed to be different from the last time, thus helping to have multiple banks to work concurrently. Furthermore, arbitration complexity is also reduced as the arbitrator only needs to select among B choices. Simulation results show 10.8% improvement in IPC and 21.0% improvement in DRAM efficiency (defined as DRAM active cycles over total DRAM cycles) of this design, with arbitration time appropriately accounted for. The improvement is greater for larger B due to the higher degree of BLP.

Although this design addresses incoming buffer blocking, arbitration, and BLP issues, it still has two drawbacks:

- While draining from different queues increases BLP, it destroys the data locality in the original program. This significantly increases miss rate (20.8% more on average).
- Memory requests that go into the same queue may have mixed (random) row and column address, thus susceptible to row buffer conflicts.

To address these issues, we need a more comprehensive, yet low-cost, reordering scheme, as proposed next.

2.4 CART: Cache Access Reordering Tree

2.4.1 Overview

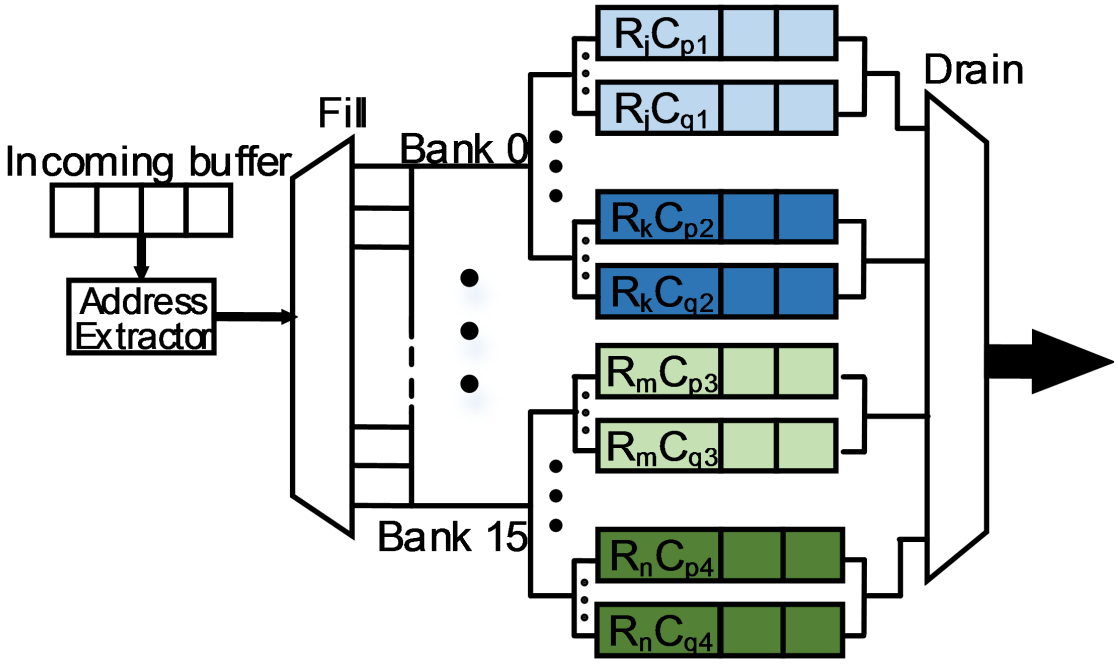


Figure 2.4: Diagram of the proposed CART.

Our objective is to reorder memory accesses at L2 cache in a cache-friendly and DRAM-friendly way. To achieve this, in addition to classifying memory requests based on bank addresses, the requests need to be further classified by row and column addresses. Ideally, requests with the same bank, row and column addresses should be grouped together, because they access the same row buffer in the DRAM and belong to the same cache line (i.e., same column). However, this grouping method is highly impractical as there are thousands of different rows in a bank

and tens of columns in a row. It is impossible to provide a separate queue for each combination of (bank, row, column). Therefore, we need a way to mimic the effects of having a large number of queues but using a limited number of physical queues. To realize this, we propose *Cache Access Reordering Tree (CART)*.

As shown in Figure 2.1, CART is positioned right before L2 cache to actively reorder memory requests. Figure 2.4 illustrates the structure within CART. Every memory request that pops out from the incoming buffer will go through a reordering tree to reach one of the FIFO *leaf queues*. To achieve a high degree of BLP, CART provides a tree *branch* for each bank (e.g., 16 branches if a DRAM chip has 16 banks). Within a branch/bank, instead of having a leaf queue for each pair of (row, column), there is a small pool of leaf queues (e.g., 8 queues). A leaf queue can be dynamically assigned to any (row, column) pair to buffer memory requests that have the matching row and column addresses. The *fill* policy determines if a memory request should be put into an existing leaf queue or be assigned a new leaf queue. The *drain* policy determines which leaf queue to output a memory request. A leaf queue is de-assigned when it is empty. A tag is attached to each leaf queue to indicate the current (row, column) assignment of the queue. As the bank address of a branch is implicitly known, the tag includes only the information of row and column, where R_x represents the row address and C_x represents the column address. To provide fairness and avoid the cases where one row uses up all the leaf queues in a branch, each row has a fixed number of assigned leaf queues. For example, if this number is 2 and the branch size is 8 leaf queues, then there are four rows in a branch, with each row being capable of buffering memory

requests for two different columns. Each leaf queue can be very small, with only a few entries per queue.

With this structure, requests are naturally grouped by rows and columns, whereas accesses to different banks are separated in different branches. These properties make it possible to address data locality, row buffer hit, and BLP issues at the same time. The carefully-designed fill and drain policies (described in following subsections) utilize the CART structure to achieve these objectives, while simplifying arbitration efforts.

Figure 2.5 exemplifies what can be achieved by the proposed CART. MRn denotes memory request n, and Bi, Rj and Ck represent the bank address, row address and column address of this request, respectively. Figure 2.5(a) shows the original order of a sequence of memory accesses (note: leftmost request occurs first in time). With the FIFO incoming buffer in conventional L2 cache designs, there are a number of places where data locality and BLP are lost. For instance, MR0 and MR4 belong to the same row in the DRAM bank. However, by the time that MR4 arrives at the DRAM, the row buffer may have been replaced by MR1's row, causing an extra row buffer conflict. Also, MR0 and MR2 belong to the same cache line and MR2 could hit in L2 without going to DRAM. However, due to MR1 that takes place between MR0 and MR2, MR0's cache line could be replaced by MR1 if they are mapped to the same position in L2. This disrupts data locality and causes MR2 to miss in the cache.

Figure 2.5(b) illustrates the access order after performing cache-friendly reordering. By switching the order of MR1 and MR2 (e.g., by filling MR1 and MR2

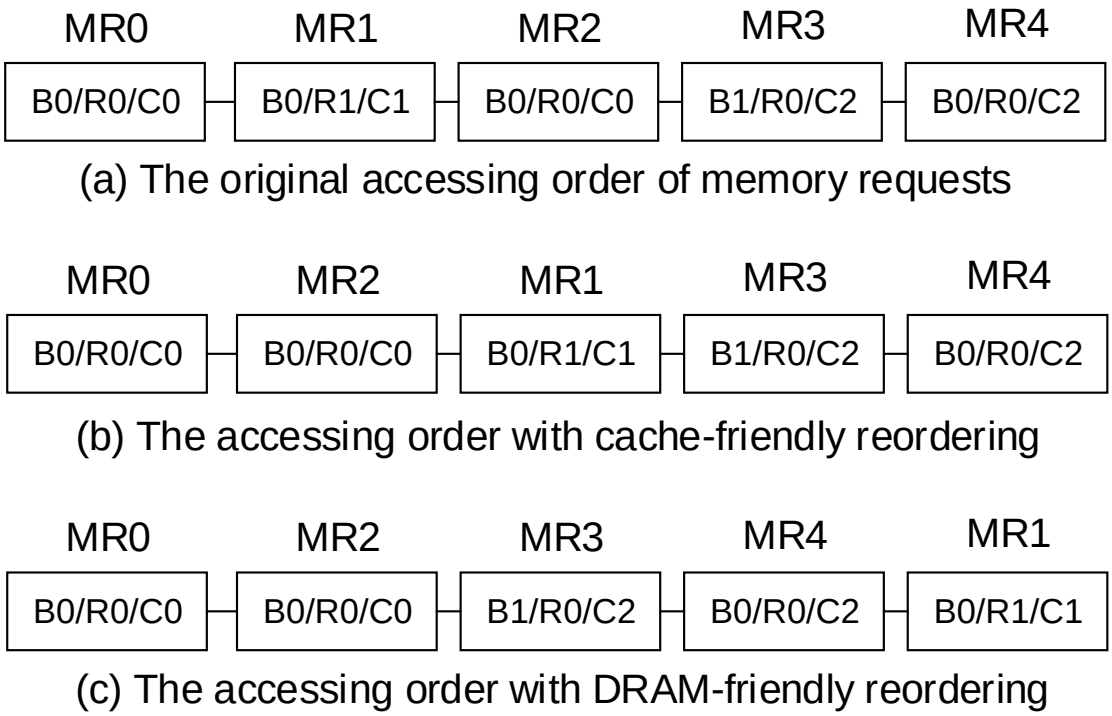


Figure 2.5: An example of the effects of CART. $Bi/Rj/Ck$ denotes that the address of the memory request (MR) is in bank Bi, row Rj and column Ck.

into different leaf queues and then drain MR0 and MR2 consecutively), MR2 can result in a cache hit without fetching from DRAM. Additionally, DRAM-friendly reordering as illustrated in Figure 2.5(c) can improve BLP and reduce row buffer conflicts. For example, MR3 has a bank address that is different from that of MR2. If MR3 and MR1 switch order (e.g., by draining different branches in CART), Bank 0 and Bank 1 can fetch the required data in parallel, increasing DRAM BLP. Furthermore, MR4 can reuse the data in the row buffer of B0/R0 if MR4 is put into one of the leaf queues that belong to B0/R0. This avoids a potential row buffer conflict.

2.4.2 Leaf Queue Allocation

A key aspect of CART is to use a small number of leaf queues to approximate the effect of having a large number of queues in a branch. The rationale behind this that, despite the tens of thousands of (row, column) address combination for a bank, there are only a limited number of (e.g., tens of) outstanding memory requests per bank. Moreover, some of the outstanding requests share the same row and column addresses, and can be merged in one leaf queue. This indicates that it is possible to use a small number of leaf queues to buffer all the outstanding memory requests, as long as there is a leaf queue or an entry in a leaf queue available by the time a new request comes.

With a given number of leaf queues in a branch, there are several queue allocation strategies. We can allocate more leaf queues to a row, so as to accommodate more memory requests for different column addresses in the row, at the cost of fewer rows. Or we can allocate fewer leaf queues to a row, which increases the number and diversity of rows in a branch, but at the cost of fewer columns per row. Additionally, under the same total buffer space, the number of entries in a leaf queue can be reduced to increase the number of leaf queues. To this end, we conducted an extensive experimental study to identify the best trade-off configuration. While details are presented in Section VI-A, we observed that the performance of providing two 2-entry leaf queues per row and four rows per branch is within 3% of the performance of a configuration that has 32 times as many buffer resources. This demonstrates the viability of using limited queues to reorder memory requests.

2.4.3 Fill Policy

The fill policy determines which leaf queue an incoming memory request should be buffered. Since CART provides one branch per bank, the fill policy only needs to select among the leaf queues in a branch. The fill policy works in a straightforward way: if one of the leaf queues contains the same row and column information as the new request, the request is merged into this queue by occupying an empty entry (in the FIFO order). If there is no empty entry in the matching queue or if there is no matching queue, a new available leaf queue is allocated to store the request, with the tag information set to the row and column addresses of the new request. Here, "available" means that an empty leaf queue is available among the leaf queues that are assigned to that row. Lastly, if no such leaf queue is available, the new memory request is stalled in the incoming buffer and retries later. Note that the probability for stalling can be kept very low with a sufficient number of leaf queues, and our study shows that the above configuration with only 8 leaf queues per branch can already achieve a near-zero probability in most cases.

2.4.4 Drain Policy

The drain policy is inherently more difficult to design than the fill policy, as the effect of a not-so-good fill decision may be delayed and partially compensated by the buffering effect of the tree, but a drain decision directly affects which requests are issued to the cache. Unfortunately, commonly-used general drain policies such as greedy, longest-first, and round-robin do not work well with CART. For example,

if the longest-first policy is applied to CART, the longest leaf queue is selected to drain in every cycle. This increases locality as all the requests in a queue share the same row and column, but squanders the opportunity for bank-level parallelism. Figure 2.6 illustrates an example. When the longest-first drain policy is applied, the first several selected requests would be in this order is Q1_MR7, Q1_MR6, Q2_MR11, Q0_MR2, Q1_MR5, Q2_MR10 (queue ID is used for tie-breaker). As can be seen, no requests are selected for Bank 2 and Bank 3, and they are not working during all this time. Similarly, round-robin policy does not work well either as it actively selects requests across different queues, thus destroying the data locality.

To address these issues, we propose a "rotating banks and same-or-longest row" drain policy to achieve both cache-friendly and DRAM-friendly order. The drain policy includes two aspects. In the first aspect, the policy selects one and only one request from a branch (i.e., leaf queues belonging to the same bank), and then immediately rotates to the next non-empty branch (bank) in the next cycle. In the second aspect, when the policy rotates back the same branch, the same leaf queue that was selected last time is selected this time, or if that leaf queue has already been fully drained, the longest leaf queue with the same row in the branch is selected (if all the leaf queues of that row are drained, simply select the longest leaf queue in the branch). Essentially, the first aspect increases BLP; whereas the second aspect increases data locality and row buffer hit, as the requests in the same leaf queue access the same cache line (if cache hit) or the same DRAM row (if cache miss). Take the example in Figure 2.6 again. Assuming the proposed policy

starts with Bank 1, since this is the first time with no prior history, the longest leaf queue, Q1_MR7, is selected. Then the policy rotates to Bank 2 and selects Q3_MR13. This is followed by Q5_MR18 in Bank 3. When the policy rotates back to Bank 1, the same leaf queue as the last time, Q1_MR6, is selected to maintain the locality, regardless of whether Q1 is currently the longest. Similar process continues, with the draining order of Q3_MR12, Q5_MR17, Q1_MR5, Q4_MR15, Q5_MR16, Q1_MR4, Q4_MR14, Q1_MR3, Q0_MR2 (because of same row with Q1), Q0_MR1, Q0_MR0, Q2_MR11, Q2_MR10, Q2_MR9, Q2_MR8. Compared with longest-first and round-robin, this order achieves substantially reduced cache misses and row-buffer conflicts while utilizing multiple banks effectively.

Table 2.1: CART design configuration.

# of leaf queue	Bank: 16; Row: 4; Column: 2
Queue size	2 entry per queue
Tag	Row + column addresses
Drain policy	Rotating banks and same-or-longest row

Table 2.2: Simulation configuration.

# of SMs	28
Warp Scheduler	GTO
Per-SM limit	48 warps, 8 CTAs
# of Memory Partitions	8
L1D cache	l16 KB, 32-set, 4-way, 32 MSHR,
Allocate on Miss, Local write-back,	
global write-through	
L2 cache	l8x128 KB, 64-set, 16-way,
32 MSHR, Allocate on Miss, write-back	
DRAM	lFR-FCFS scheduler, GDDR5, 16 banks
SM/L2/DRAM clock	1400/700/1150 MHz

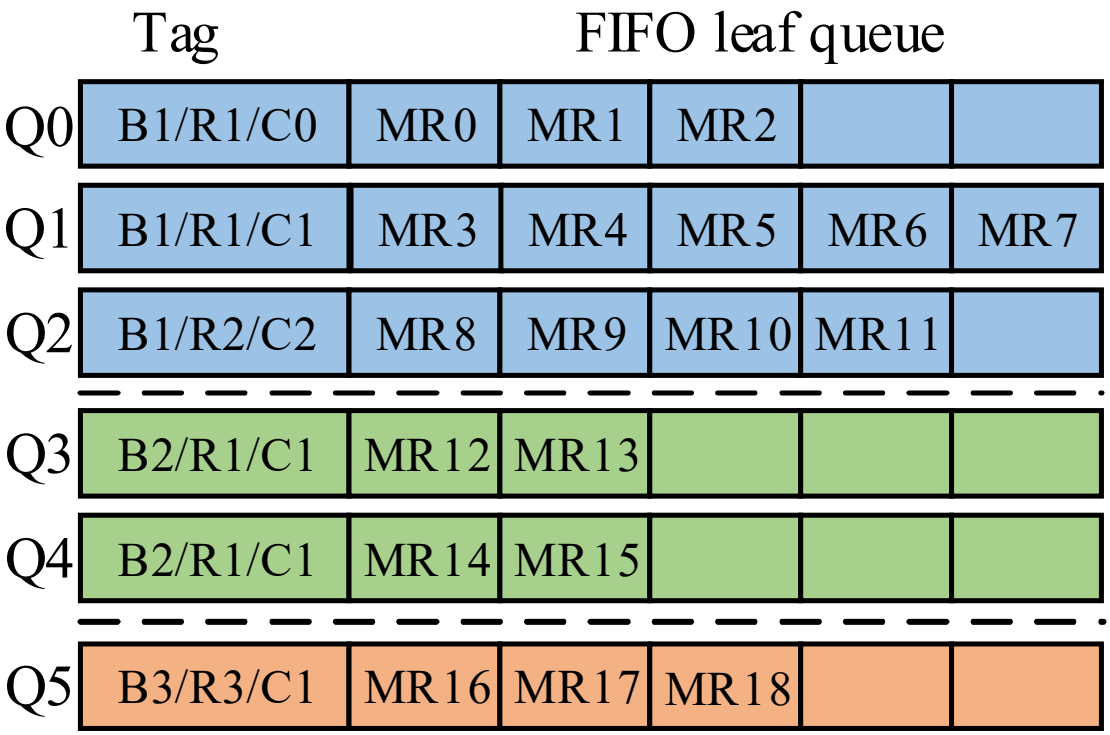


Figure 2.6: Illustration of drain policies for a given tree status.

Table 2.3: Evaluated benchmarks.

Benchmarks	**Abbr.**	**Inst./L2 Miss**	**Type**	**Benchmarks**	**Abbr.**	**Inst./L2 Miss**	**Type**
Backprop	BP	1718	C	ConvolutionTexture	CT	3104	C
Bfs	BFS	62	M	FastWalshTransform	FWT	641	M
B+tree	B+T	1497	M	QuasirRandomGenerator	QG	2798	C
Cfd	CFD	404	M	RadixSortThrust	RST	425	M
Dwt2d	DW	530	M	SortingNetwork	SN	1160	M
Heartwall	HW	5189	C	Spmv	SP	568	M
Nw	NW	198	M	Stencil	STC	614	M
Kmeans	KMS	19236	C	Sgemm	SG	9957	C
Cutcp	CUT	400509	C	Transpose	TRP	400	M
Sad	SAD	2733	C	Mri-q	MRI	120455	C
Tpacf	TP	842408	C	Scan	SCN	233	M
AlignedTypes	AT	238	M	Lbm	LBM	176	M
AsyncAPI	AA	416	M	MergeSort	MS	1436	M
BlackSchole	BS	476	M	Histo	HIS	751	M
ConvolutionSeparable	CS	701	M	SobolQRNG	SQ	1247	M

2.5 Evaluation Methodology

We implement the proposed schemes in a cycle-accurate simulator, GPGPU-Sim 3.2.2 [10]. GPUWatch [58] is employed to evaluate the energy consumption of our proposed scheme and baseline architecture. Table 2.2 lists the configuration used in the simulator. The benchmarks from Rodinia [22], Parboil [91], and Nvidia GPU Computing SDK are evaluated. Table 2.3 lists more details of the benchmarks. The second and fifth columns in the table illustrate the total number of instructions executed by the entire SMs over the number of L2 cache miss. These values reflect the extent to which the performance of the benchmarks depends on cache performance [89, 56]. The benchmarks whose total executed instructions per L2 miss are less than 1500 are considered as the memory intensive benchmarks and are marked *M* in the "Type" column. Other benchmarks whose values are over 1500 are compute intensive benchmarks and are marked *C* type.

We compare CART against the baseline architecture, as well as two state-of-the-art techniques, MRPB [47] and RACB [27]. MRPB uses memory request prioritization buffer to reorder memory requests in the L1D cache and bypass selected requests. RACB uses bypassing technique in both L1D and L2 cache according to the resource availability in these caches. Note that all the schemes employs the widely used FR-FCFC scheduling [85] at the memory controllers. Therefore, memory request reordering opportunity before DRAM is exploited in all the schemes.

2.6 Results and Analysis

2.6.1 Exploring CART Design Space

While CART works better with more leaf queues, it may not necessary to provide a large number of queues, particularly with cost consideration. To gain insight on how to identify a good trade-off design between the resource consumption and performance improvement, we have examined the impact of different leaf queue numbers and queue sizes on performance. Since the number of branch is fixed to one branch per bank, we only need to explore the design space of row numbers, column numbers and queue sizes (entry numbers). Figure 2.7 plots the impact on performance improvement over baseline architecture for several memory-intensive benchmarks by using various configurations. nR denotes that there are at most n different row addresses in the leaf queues belonging to each bank. nC means that for the leaf queues belonging to a specific bank and row address, there are at most n different column addresses of memory requests; the nE represents the entry numbers in each FIFO queue.

It can be seen that, more resource leads to higher performance improvement for CART. However, the difference is not very large, demonstrating that CART does not need an impractical number of queues for each row and column combination. Figure 2.7 shows a diminishing return when adding more resources. In fact, the average performance improvement drops only by 2.0% when the resource for CART is reduced from 8R/8C/8E to 4R/2C/2E, which is a resource reduction of 96.9% (512 entries vs. 16 entries). While not shown in the figure, providing an extremely

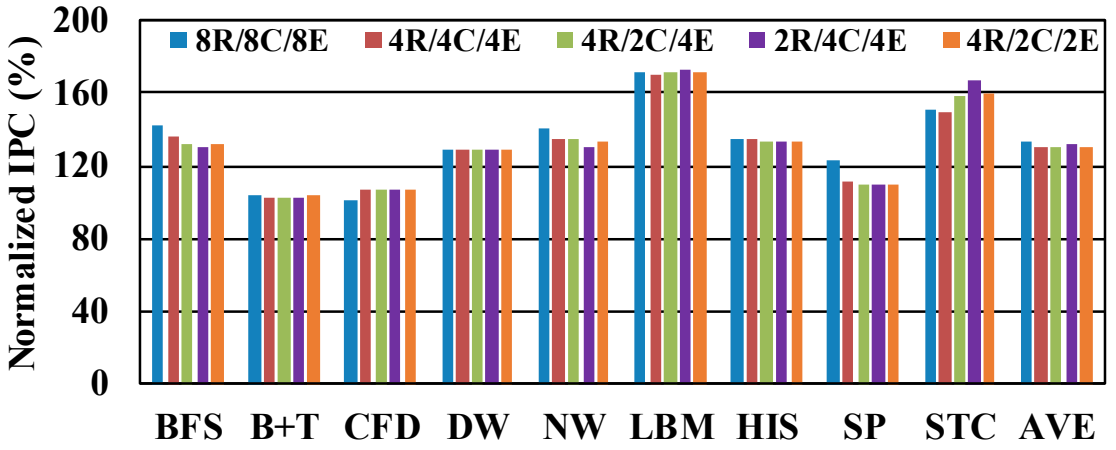


Figure 2.7: Finding good performance-cost tradeoff for CART.

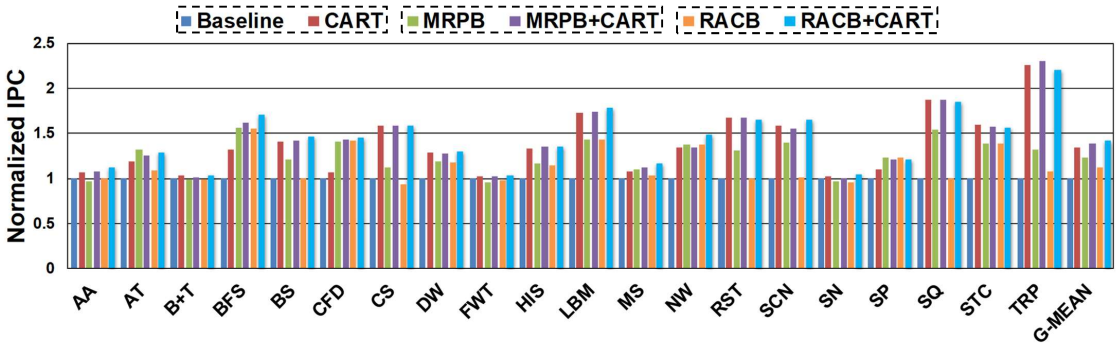


Figure 2.8: Performance comparison of different schemes for memory-intensive benchmarks.

large number of leaf queues and entries (approaching ideal performance) has a performance improvement within 3% of the 4R/2C/2E configuration. Based on this study, we select 4R/2C/2E as the current configuration of CART in our further evaluation. The table 2.1 summarizes the configuration of CART.

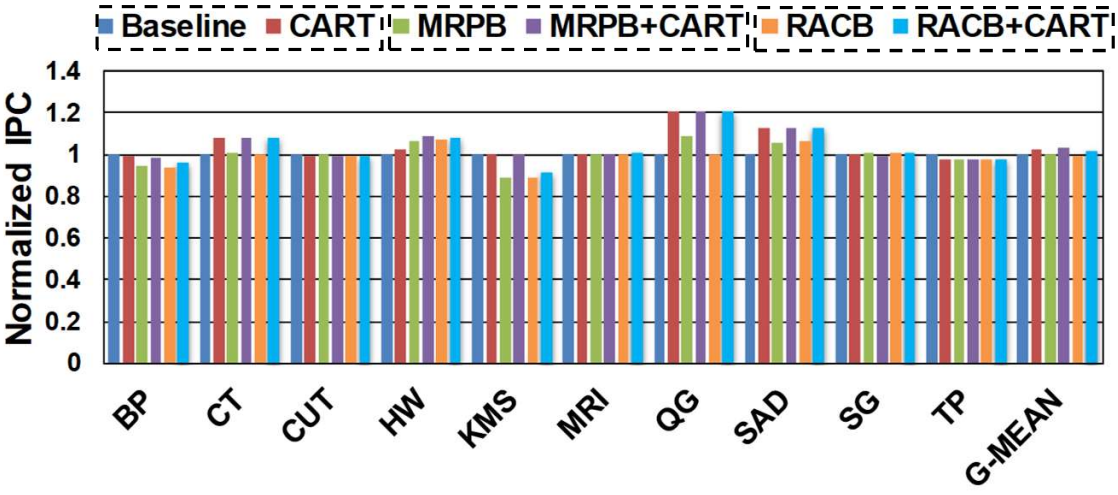


Figure 2.9: Perf. comparison for compute-intensive benchmarks.

2.6.2 Performance Comparisons

Our performance comparison consists of two main parts. The first part is to evaluate how effective the proposed CART is when applied alone, compared with other schemes (baseline, MRPB and RACB) applied alone. The second part is to evaluate if CART can complement other schemes to improve the efficiency of memory subsystem. As with other works in the area [89], we plot the results of memory-intensive and compute-intensive benchmarks separately.

First of all, Figure 2.8 shows the overall performance improvement of different schemes for the memory-intensive benchmarks. Compared with the baseline, the proposed CART alone can achieve 34.2% average IPC improvement (geometric mean). For some benchmarks, such as Transpose (TRP), CART even achieves 2.26X IPC improvement. This shows that actively reordering the incoming requests to a cache-friendly and DRAM-friendly order can help relieve the pressure on mem-

ory subsystem. MRPB mainly works on the L1D cache efficiency by reordering the cache in a cache-friendly order and bypassing upon associativity-stalled requests. MRPB improves the average IPC of memory-intensive benchmarks by 23.4%. In addition, RACB focuses on the resource-aware L1D and L2 cache bypassing. When the resource in L1D or L2 cache is no longer available, the bypassing is activated. The average IPC in RACB increases by 12.6% over the baseline. Therefore, the proposed CART performs better than the other two schemes by a large margin.

Second, as the proposed CART improves the efficiency of both L2 cache and DRAM, the combination of CART and MRPB or RACB can explore the benefits across L1D, L2 cache and DRAM. Figure 2.8 also shows the performance improvement of combined schemes. The combination of CART and MRPB can improve the average IPC by 38.6%, and the combination of CART and RACB can achieve average IPC improvement by 41.5%. Those two improvements are much higher than applying MRPB and RACB alone. This illustrates that the proposed scheme exploits an new opportunity that is largely complementary to existing ones.

We also examine the effect on compute-intensive benchmarks by using different schemes. The results are shown in Figure 2.9. The performance improvement of MRPB and RACB are both within the 1.0%. CART is slightly better with an average improvement of around 2.5%, although some benchmarks have observed larger improvement (e.g., 20.8% in QG and 12.7% in SAD). These results on compute-intensive benchmarks are understandable as they are insensitive to memory. For a fair comparison, when considering all the memory-intensive and compute-intensive benchmarks together, CART is still able to achieve an average

improvement of 26.5%.

2.6.3 Insight of Performance Improvement

Several factors may contribute to the performance improvement of CART: reduction of row buffer conflicts, improvement in L2 hits, and increase of bank utilization. The proposed CART pursues the benefits of more row buffer hits by giving a high priority in the drain policy to the memory requests that have the same row addresses as the previously accessed row. Figure 2.10a compares the number of row buffer conflicts in the baseline architecture and CART. Many memory intensive benchmarks are observed to have a reduction of row buffer conflicts. On average, the benchmarks with CART have 12.3% decrease in row buffer conflicts. This decrease helps to reduce the time in replacing the content of row buffers, thereby increasing DRAM efficiency. Similarly, Figure 2.10b and Figure 2.10c show the impact of CART on L2 cache hit and by bank utilization, respectively. Note that, while not all the benchmarks have improvement in all the three aspects, we have observed and verified that each benchmark has large improvement in at least one of the aspects.

2.6.4 Hardware Implementation

We use Cacti 6.5 [99] and RTL implementation to estimate the area and timing of CART. All the key components of CART are implemented and evaluated for

hardware cost, including the SRAM-based leaf FIFO queues, the address extractor, the comparators in leaf queues (eight parallel comparators to allow an incoming request to be selected and written into a leaf queue in each cycle), the request selector to reflect the drain policy, etc. With all the components together, CART incurs 0.016 mm^2 per L2 cache partition under 45 nm, which is only 1.7% relative to each L2 cache partition. In comparison, MRPB adds 10.5% hardware overhead (also relative to L2).

2.6.5 Energy efficiency

Due to the small hardware overhead, the proposed CART has very limited overhead on power consumption. As a result, the overall energy consumption of the GPU is reduced due to the shortened execution time. We lumped the CART power overhead in GPUWattch, and simulation results show that CART reduces the energy consumption by 18.9%, compared with the conventional design.

2.7 Related Work

Cache bypassing: Several studies have focused on cache bypassing to alleviate cache pressure for GPUs. Xie et al.[103] use compilers to analyze cache utilization of the code based on specific metric and select related instructions for cache access and bypass. Dynamic cache bypassing is also proposed [27, 59, 104, 47]. Besides using compilers, Xie et al.[104] propose to bypass memory requests in a thread

block based on the ratio of thread blocks that cache or bypass at runtime. A data locality monitoring mechanism is developed by Li et al.[59] to select highly reusable data to be stored in L1D cache. Jia et al.[47] and Dai et al.[27] both use resource aware technique to bypass memory requests that are stalled in cache to increase memory efficiency. However, all those cache bypassing schemes do not consider the possible impact on DRAM, whereas our work increases DRAM working efficiency by improving DRAM BLP and reducing row buffer conflicts during filling and draining.

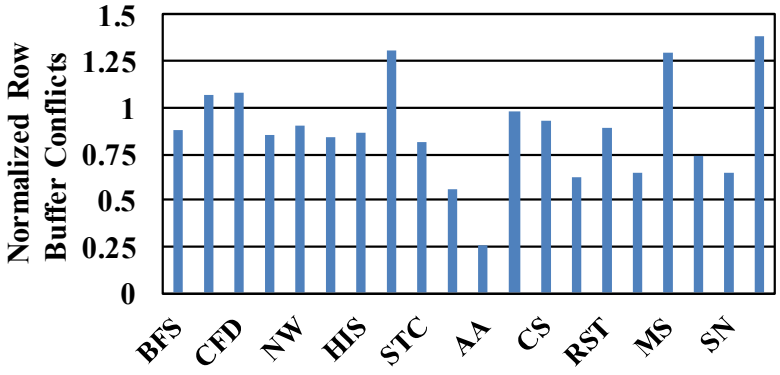
Warp scheduling: The memory efficiency can be also improved by optimizing the warp scheduler (e.g., [71, 86, 89]). We evaluated several warp schedulers (GTO, LRR, two-level). The proposed CART achieves similar relative improvement for different warp schedulers, indicating that CART is not sensitive to warp scheduling. It might be that the effects of warp scheduling on the access order has been degraded (filtered) by the L1 cache before L2.

Data-locality optimizing: Besides passively bypassing memory requests, an active optimization, MRPB (memory request prioritization buffer [47]), is proposed that actively reorders the memory requests in the L1D cache to a cache friendly order to increase cache hits. Also, a memory access scheduling policy is proposed [69] to reduce the negative impact brought by inter-thread interference. This improves the throughput of DRAM. As our proposed CART aims to change the memory requests to a cache-friendly and DRAM-friendly order before entering L2 cache, CART can be used to complement those schemes, as exemplified in evaluation.

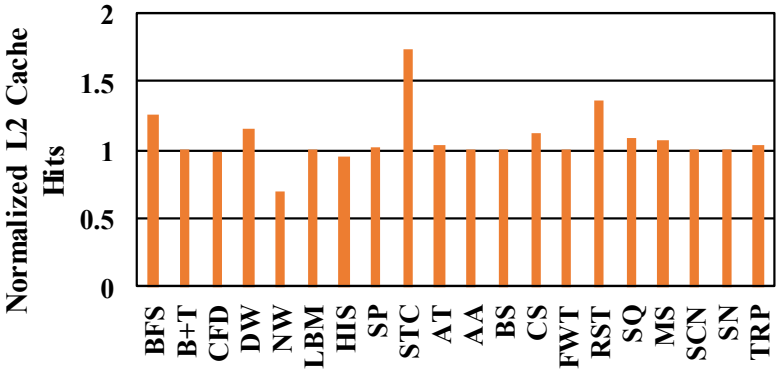
Software-level: Software-level schemes can also be used to improve GPU memory subsystem. Streamline is proposed [106] to resolve irregular memory references and control flows through data reordering and job swapping in software. Another work [100] proposes a new algorithm to minimize non-coalesced memory accesses. However, requests to L2 may come from different SMs, thus revealing new reordering opportunity. This is exploited in our scheme that reorders memory requests before entering L2 cache. It is completely done in hardware without the need for application profiling.

2.8 Conclusion

The order of memory accesses plays a significant role in determining the efficacy of memory subsystem. In this work, we propose Cache Access Reordering Tree (CART), a novel architecture that actively reorders memory requests in L2 cache to relieve the congestion of L2 and to increase DRAM working efficiency. The proposed CART is able to improve the IPC of a wide range of memory-intensive workloads by 34.2%. The results also show that the proposed scheme can complement other memory subsystem optimization techniques to further improve system performance.



(a) Reduction of Row Buffer Conflicts.



(b) Improvement of L2 Cache Hits.

(c) Increase of Bank Utilization.

Figure 2.10: More details on Performance Improvement.

Chapter 3: Dynamically Linked MSHRs for Adaptive Miss Handling in GPUs

3.1 Basic Idea

In this chapter, we propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that allocates miss handling resources flexibly and adaptively to meet the diverse miss behaviors of applications. In DL-MSHR, entries are formed dynamically from a pool of available slots. A slot can be assigned as an independent entry for processing a primary miss, or can be linked after another slot in an existing entry to increase the capacity of processing secondary misses. The number of slots that each entry has depends on the frequency of memory accesses to the corresponding cache line. This approach self-adapts to primary-miss-predominant applications by forming more entries with fewer slots, and adapts to secondary-miss-predominant applications by having fewer entries but more slots per entry. We also propose four additional optimization techniques to further increase the efficiency of DL-MSHR. Evaluation results show that, compared with conventional MSHRs, the proposed DL-MSHR is able to reduce the primary- and secondary-miss-induced reservation fails in MSHRs by 88.1%, improve the MSHR utilization by 53.7%, and increase the overall IPC by 19.2% with only 0.6% and 0.1% area overhead on L1D and L2 cache, respectively. Moreover, DL-MSHR can comple-

ment existing techniques and achieve an additional 26.3% IPC improvement on top of MRPB (Memory Request Prioritization Buffers) [47]. The average IPC of DL-MSHR is even 8.0% higher than the conventional MSHR configured with 4X the amount of hardware, i.e., doubling the number of entries *and* doubling the number of slots per entry.

3.2 Background

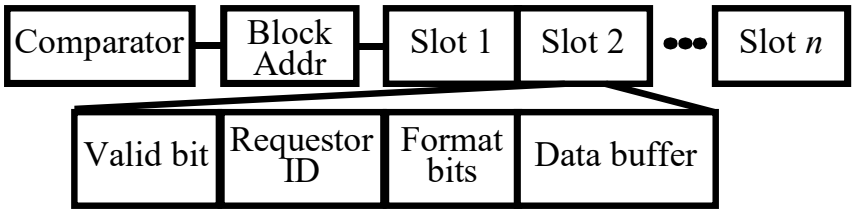
Miss handling architecture (MHA) is critical to memory level parallelism and system performance, as MHA feeds and tracks concurrent miss requests that are issued to the next level of memory hierarchy. Over the years, miss handling architecture has been evolving continuously and has unlocked an increasing amount of parallelism that can be achieved by cache and memory. This section explains several key considerations of MHA that lead to the explicitly-addressed, MSHR-based MHA design today.

Lockup cache vs. lockup-free cache. When cache was originally introduced, the associated MHA can handle only one outstanding miss at a time (i.e., lockup cache). To read a data, the data address is used to search the cache. A cache hit returns the requested data right away; whereas a cache miss requires the MHA to first record the pertinent information of the request and then issue the request to the next level in the memory hierarchy. Before the data is back, the cache does not process new misses and is "locked up". Writing data is similar (as most cache designs use write-on-allocate policies), except that the MHA needs to

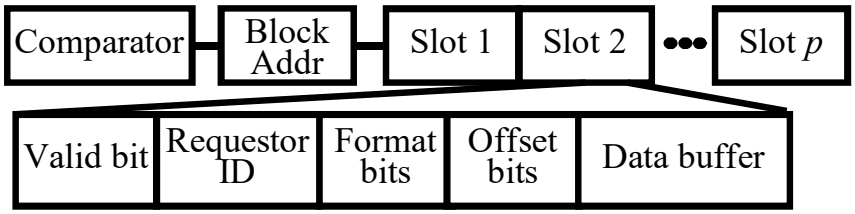
provide a data buffer to store the new data temporarily before the corresponding cache line is allocated and available.

To support lockup-free caches, multiple *Miss Status Holding Registers* (MSHRs) are added to the MHA to keep track of multiple outstanding misses concurrently[55]. Each cache miss is allocated one MSHR entry which records the information of the miss, such as the requester ID, cache block address, requested data tag, new data for write-back (in case of writing), etc. Once the requested data is returned, the data can be forwarded back to the corresponding requester based on the information retrieved from the MSHR. The cache can accept new misses from processing cores as long as there are free MSHRs available.

Primary miss vs. secondary miss. As the smallest unit for data transferring between two cache levels is a cache line rather than individual words, additional optimization opportunity exists in combining multiple data requests to the same cache line. To exploit this opportunity, cache misses are divided into two types. A *primary* miss occurs when the cache line containing the requested word does not exist in the cache and a new MSHR entry needs to be allocated. A *secondary* miss occurs when the requested word shares the same cache line of an outstanding miss, in which case no new request needs to be issued to the next level since the requested cache line is already on the way. Note that the requested word in the secondary miss could be to a different word in the same cache line, or to the same word as the primary miss but from a different requester (e.g., a different core). To accommodate this, each MSHR entry further consists of several slots to keep track of individual word requests. An address comparator is associated



(a) Implicitly addressed MSHR (n=cache line size/word size)



(b) Explicitly addressed MSHR (p can be smaller than n)

Figure 3.1: Implicitly and explicitly addressed MSHRs.

with each MSHR entry to check if any incoming cache miss shares the same block address of the cache line that the MSHR is allocated to. if yes, a free slot in the matching MSHR entry is be needed; if not, a free MSHR entry would be needed. The comparison is done in parallel across all the MSHRs, similar to a fully associative cache. In this chapter, we use the term *entry-full* to refer to the case where no free MSHR entry is available, and use *merge-full* to refer to the situation where no free slot is available within an MSHR entry.

Implicitly vs. explicitly addressed MSHR. To realize the functionality of MSHRs in hardware, Kroft proposes an implementation based on implicitly addressed MSHRs [55]. As depicted in Figure 3.1a, in this architecture, an MSHR entry provides a pre-allocated slot for each addressable word in a cache line. All the slots share the same block address, but the offset bits within a block (to spec-

ify each word) do not need to be recorded in a slot (i.e., the words in a block are implicitly addressed by using the position of the slots). If a particular word in a cache line is requested, the requester ID and other related information is recorded in the corresponding slot. As each word in a cache line has exactly one slot, an MSHR entry is able to track multiple secondary misses, provided that they request different words in a cache line. However, secondary misses to the same outstanding word are denied because there is no place to store more than one copy of tracking information. Although this design has simple control, having at most one outstanding miss per word is a very severe limitation, especially in the many-core era. Moreover, reserving one slot per word may lead to very low efficiency of MSHR given the large cache line size in contemporary processors (e.g., 32 words).

To overcome the drawbacks of the above design, Farkas and Jouppi [30] propose the explicitly addressed MSHR. As illustrated in 3.1b, the number of slots, p, in an MSHR entry does not have to be the same as the number of words in a cache line (p is the same across all the MSHRs). Instead, every slot is generic and can be used to track any word in the cache line. Consequently, the offset of the word needs to be explicitly recorded in the slot. Although the offset requires additional bits, the achieved savings in the reduced slots and the benefits of increased flexibility far outweigh the overhead, which makes this design the *de facto* MHA in most of the current commercial processors including Core i7[26], Xeon E5[102], and GTX960[74].

While the previous discussions mainly focus on L1 cache for primary and secondary misses, similar situations also exist in L2 cache, but at the granularity of

cache lines (instead of words). A cache line in a shared L2 may be accessed by multiple private L1 caches in different cores, thus requiring a multi-slot L2 MSHR entry to track these requests. For example, a private L1 may request a cache line from L2. If the line is not present in the L2, an L2 MSHR entry is allocated to track this primary miss while the line is being fetched from the memory. Meanwhile, if another private L1 cache has a write request to L2 for the same line, this request is allocated another slot in the above entry (i.e., secondary miss), and the write data from the write request is temporarily stored in the data buffer of that slot [1]. The explicitly addressed MSHR design also works more efficiently than the implicit one for L2, as there is no need to provide a reserved slot for each L1. Note that, if any of the read and write request results in the replacement of a dirty line, the dirty line does not need a MSHR slot; instead, it is placed into the evicted buffer and later written back to the memory.

3.3 Motivation

The success of the explicitly addressed MSHR design demonstrates the importance of having an efficient miss handling architecture for enabling memory-level parallelism. However, this architecture may no longer be sufficient in handling the increasing diverse application miss behaviors.

[1] There can be multiple read and write requests to the same cache line in L2. To ensure correctness (e.g., consider the sequence of W1, R1, W2, R2), the write data from different private L1 cache requests cannot be combined in an L2 MSHR entry, thus requiring each slot to have a data buffer.

```
// Kernel in BlackScholesGPU
1: __global__ void BlackScholesGPU(
2: float *d_CallResult,float *d_PutResult,
3: float *d_StockPrice, float *d_OptionStrike,
4: float *d_OptionYears, float Riskfree,
5: float Volatility, int optN 6: ) {
7: int tid = blockDim.x * blockIdx.x + threadIdx.x;
8: int THREAD_N = blockDim.x * gridDim.x;
9: for(int opt = tid; opt < optN; opt += THREAD_N)
10: BlackScholesBodyGPU(
11:       d_CallResult[opt],
12:       d_PutResult[opt],
13:       d_StockPrice[opt],
14:       d_OptionStrike[opt],
15:       d_OptionYears[opt],
16:       Riskfree,
17:       Volatility
18: ); }
```

Figure 3.2: Blackscholes (primary-miss-predominant).

3.3.1 Diverse Application Cache Miss Behaviors

We first characterize applications by examining whether their predominant misses are primary or secondary misses. The results can help to understand the diverse demands on miss handling architecture. While several works have studied GPU workloads in detail, to our knowledge, no research has examined from the perspective of cache miss types, as defined below.

Primary-Miss-Predominant Applications. This type of applications exhibit a high demand for MSHR entries but not the slots within an entry. As an example, Figure 3.2 shows the kernel of the *blackScholes* benchmark from the NVidia SDK [76] that is primary-miss-predominant. For this kernel, there are 7

```
// Kernel in Aligned Types
1: template<class TData>__global__ void testKernel(
2: TData *d_odata, TData *d_idata, int nE)
3: {
4: int tid = blockDim.x * blockIdx.x + threadIdx.x;
5: int numThreads = blockDim.x * gridDim.x;
6: for(int pos = tid; pos < nE; pos += numThreads)
7:       d_odata[pos] = d_idata[pos];
8: }
```

Figure 3.3: AlignedType (secondary-miss-predominant).

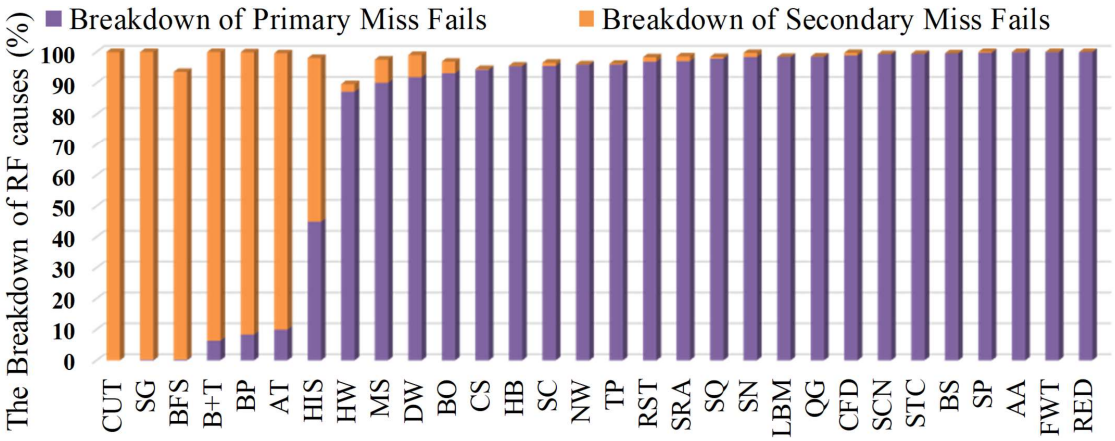


Figure 3.4: Breakdown of reservation fail (RF) causes.

different floating variables from line 11 to 17 (4 bytes each) that need to be loaded before further calculation. If running on a GTX750Ti using all the 640 CUDA cores simultaneously (5 streaming multiprocessors (SMs) $\times$ 128 cores/SM = 640), up to 140 cache lines (640 $\times$ 7 variables $\times$ 4 bytes / 128 byte/line = 140, assuming perfect coalescing) could be requested in a cycle which, theoretically, needs 140 MSHR entries to track the information. Hence, the 32 entries of MSHRs in GTX750Ti are very easy to cause execution stall. The primary-miss-predominant

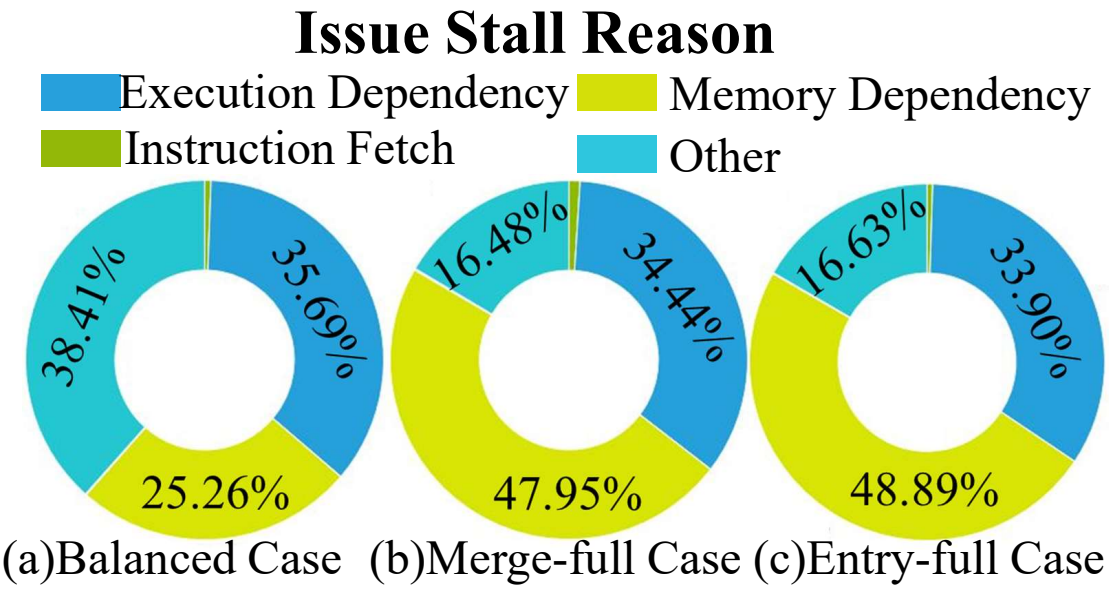


Figure 3.5: Percentage of execution stall reasons.

applications can significantly benefit from an increase in the number of MSHR entries.

Secondary-Miss-Predominant Applications. Applications in this category have a high demand for the slots in MSHR entries but less so for MSHR entries. Figure 3.3 shows the kernel of the *alignedTypes* benchmark involving array operations. Array elements are usually stored sequentially in the address space. When multiple threads are executing this kernel simultaneously, these threads may likely access the same cache lines with good spatial locality and high number of secondary misses. Assuming the floating type for the TData template in line 1, there can be up to 128 byte/line / 4 bytes = 32 requests, which greatly exceeds the 8 slots in each MSHR entry in GTX750Ti. To increase the capacity of handing secondary misses, more slots have to be added. In the current MSHR architecture,

this is very costly as 1) each slot contains a data buffer for the possible write miss, and 2) every MSHR entry has the same number of slots, so even adding one slot per entry would considerably increase hardware overhead.

We studied a number of applications from the NVidia SDK [76], Rodinia [22, 23] and Parboil [91] benchmark suites. Figure 3.4 presents the breakdown of reservation fails resulted from primary misses and secondary misses (other sources of RFs account for less than 3%). GPGPU-Sim [10] with a typical configuration (more details in Section 5.3) is used to simulate the benchmarks. The figure exhibits a great diversity, with some applications such as *b+tree* and *bfs* having reservation fails predominately from secondary misses, and applications such as *blackSholes* and *scan* having reservation fails predominately from primary misses. These results indicates that a static, one-size-fits-all MSHR architecture may not be the most efficient design to handle diverse GPU workloads.

To verify the merge-full and entry-full phenomenons in MSHR are not synthetic issues of the simulator, we have developed a microbenchmark that tests the MSHR of a recent GPU. Our intention is not to expose the publicly unavailable MSHR details of real GPUs, but rather to show that merge-full and entry-full indeed create performance issues in recent GPUs. There are three kernels in this microbenchmark that represent a balanced case, a merge-full case, and a entry-full case, respectively. Each kernel consists of 64 blocks with 256 threads per block, totaling 16384 threads. The balanced case has relatively balanced primary and secondary misses to cache. In the merge-full case, half of the threads access the same cache line, which causes a large number of secondary misses. In the entry-full case, the threads occupy

different cache lines (i.e. primary misses) as much as possible while minimizing secondary misses. A GTX960 graphics card is employed to execute the three kernels, and the NVidia Nsight tool [73] is used to collect the stalling data of the GPU. Figure 5 compares the percentage of various reasons that cause execution stall of the tested GPU. In the doughnut chart generated by Nsight, the percentage of stall from "memory dependency" increases from 25.3% in the balanced case to around 48% in the merge-full and entry-full cases, highlighting the severe negative impact of MSHR merge-full and entry-full behaviors. This is particularly evident in the merge-full case where half of the threads access the same cache line. One might expect that such access pattern would lead to a large number of cache hits and reduced data stall. However, the limited slots in each MSHR entry causes frequent merge-full situations and prevents further data requests from being serviced by the cache and MSHRs. While the performance impact of other benchmarks may not be as large as our microbenchmark, the experiment here demonstrates that the MSHR issue indeed exists in current practice.

3.3.2 Need for Dynamic Miss Handling

Under current MSHR architecture, addressing diverse application miss behaviors needs to increase the number of MSHR entries and slots. Note that both entries and slots need to be increased. Missing either of the two aspects would result in a class of applications to suffer from primary miss induced or secondary miss induced reservation fails. This approach is costly and inefficient for two major

reasons. First, MSHRs are implemented as content-addressable memory (CAM). Each MSHR entry has an address comparator, and all the entries need to be searched in parallel up on a cache miss. This places a high capacitive load at the output gate of the upstream address decoders. Our evaluation based on CACTI 6.5 [68] confirms that the search delay and area cost of MHA rise superlinearly as the number of entries increases. However, these overhead is relatively small if the total number of entries is not large (i.e., the negative effect of superlinear growth becomes substantial *only* when the base number is large).

Second and more importantly, each slot contains a data buffer to temporarily store write-back data in case of a write miss. Thus, increasing the number of entries and/or slots would substantially increase the overall area of the MHA. For example, as shown in Section VI, doubling the number of entries and slots for L2 MSHR incur an area overhead of 22.3% in terms of L2 cache area, and 33.4% power overhead. Nevertheless, the performance gains from this are still very limited. Clearly, directly increasing the size of MSHR is not a cost-effective solution.

This calls for a flexible and dynamic MSHR design that can utilize hardware resources smartly. The opportunity comes from the fact that primary-miss-predominant applications need a large number of entries, but only few slots within each entry is occupied. Similarly, secondary-miss-predominant applications have high demand for the slots within certain entries, but many other MSHR entries (and their slots) may still be free. This opportunity is exploited in the approach proposed in this chapter.

3.3.3 Other Related Work

GPU architecture has been improved from various aspects (e.g., [48, 54, 77, 57, 93, 41] and many others). However, only a few works have targeted the efficiency of miss handling architecture. To reduce cache look-up time and increase bandwidth, Tuck *et al.* [94] propose a hierarchical MHA, where a small MSHR file is provided at each cache bank to process the majority of secondary cache misses, and a large MSHR file that is shared by all the banks to handle long latency misses. In addition, Jahre *et al.* [45] propose to shrink the miss handling bandwidth for a specific core that delays the execution speed of other cores, thereby achieving a higher overall speedup. Neither of the above two designs can dynamically adjust the number of MSHR entries or slots that can be self-configured to best suit the needs of applications as proposed in this work. Loh[63] proposes Vector Bloom Filter (VBF) that can provide faster access for large MHA and can dynamically shrink MSHR capacity. However, VBF does not explore the opportunity in utilizing the unused slots in an entry that is already allocated to a primary miss, whereas DL-MSHR utilizes these slots by decomposing each entry into slots and dynamically linking them. In evaluation, we compare DL-MSHR with a perfect VBF where the MHA access time is one cycle regardless of the MHA size. In addition, Power *et al.* [79] propose a region-based coherence to reduce MSHR entries in the coherence directory in heterogeneous systems, and Qureshi *et al.* [82] propose a linked structure in V-Way cache to reduce the unbalanced set access problem. Both works are related but have very different contexts than this work.

Also closely related to MHA are reservation fails, which may occur due to several reasons such as MSHR entry-full, MSHR merge-full, miss queue full, cache reservation full etc[52]. If the data request at the head of the request buffer to cache encounters a reservation fail, subsequent requests will be blocked even though they could have been processed in three cases: 1) the reservation fail is caused by entry-full with no available MSHR entry to track this primary miss, but subsequent requests could have been merged into other allocated MSHR entries (i.e., secondary misses); 2) the reservation fail is caused by merge-full with no slot to accept this secondary miss in a particular MSHR entry, but subsequent requests could have been assigned with other available MSHR entries; and 3) the blocked subsequent requests could have hit in the cache and thus do not need MSHRs.

Several works have been proposed to address reservation fails in some degree. Jia *et al.*[47] and Dai *et al.*[27] both use resource-aware cache bypassing techniques to bypass memory requests when they suffer stall in the cache. Xie *et al.*[103, 104] propose a compiler level cache bypassing technique. The compiler analyzes the cache utilization of a program based on the developed metric, and then selects certain instructions to access or bypass cache. While these cache bypassing techniques are effective in avoiding reservation fails when they are imminent, they do not explore the opportunity in improving miss handling architecture to reduce the likelihood of reservation fails in the first place. Another technique MRPB[47] is also proposed to actively reorder the requests into a cache-friendly order before accessing L1D cache and the associated MSHRs. Nevertheless, the effectiveness of MRPB is greatly limited by the "static" nature of MSHRs, e.g., when MSHR

is entry-full (but not all the slots in the entries are occupied), no primary miss can be accepted even if these primary miss requests are perfectly reordered. Our proposed scheme addresses this issue by dynamically forming MSHR entries and slots from a pool of unified resources, thus complementing MRPB in a different way as shown in evaluation.

3.4 Dynamically Linked MSHR

3.4.1 The Basic Idea

We propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that allocates miss handling resources flexibly and adaptively. The basic idea is to decouple the static binding between a conventional MSHR entry and its constituting slots. Each DL-MSHR entry is dynamically formed from a pool of available slots. The adaptivity of DL-MSHRs is reflected in two aspects. Across applications, more entries with fewer slots are formed to meet the demand of primary-miss-predominant applications, whereas fewer entries but more slots per entry are formed for secondary-miss-predominant applications. Within an application, the number of slots that each entry has can also adapt to the frequency (demand) of memory accesses to the corresponding cache line. This flexibility allows DL-MSHRs to satisfy some extreme primary and secondary miss demands without the need for more physical entries or slots.

Figure 3.6 shows how DL-MSHRs integrate with other components of the sys-

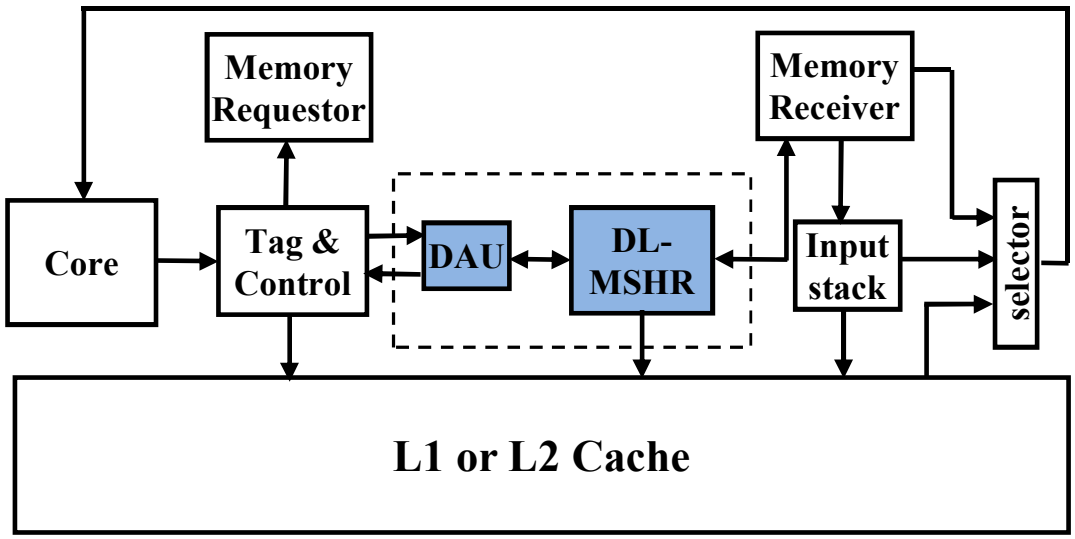
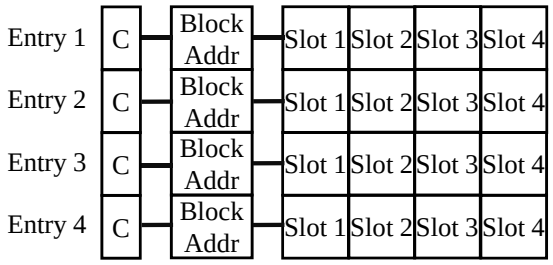
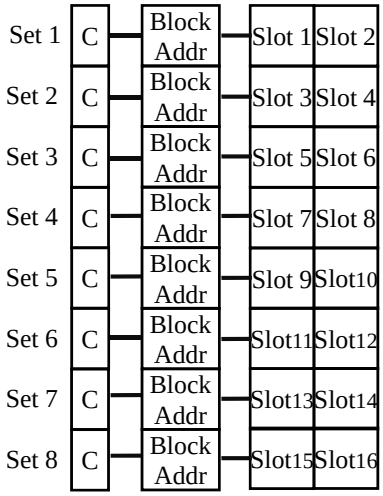


Figure 3.6: Overview of dynamically linked MSHRs (the new and modified components are highlighted).

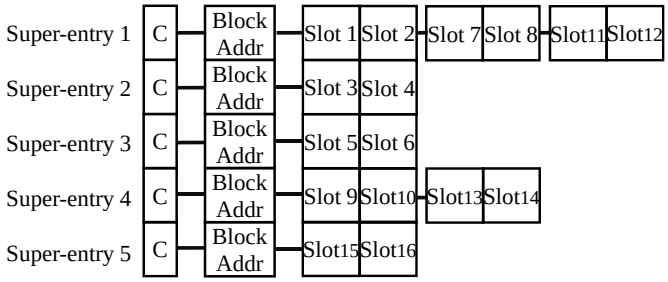
tem. At the high level, an array of DL-MSHRs replaces the conventional MSHR array to track multiple outstanding misses. A *Dynamic Allocation Unit (DAU)* is developed to control the operations of DL-MSHRs. The DAU is placed between the original Tag & Control unit and the DL-MSHR array. Upon a read or write request from the processing core or from the previous level in the memory hierarchy, the Tag & Control unit checks if the request hits in the cache. If not, the Tag & Control tries to insert the request to an MSHR and, if successfully (receiving acknowledgement from the MSHR), issues the request to the next memory level. With DL-MSHR, the DAU intercepts the signals from the Tag & Control and inserts the request to a dynamically linked DL-MSHR slot.



(a) Conventional MSHRs



(b) Physical view of DL-MSHRs



(c) A possible logical state of DL-MSHRs

Figure 3.7: Illustration of conventional MSHRs and dynamically linked MSHRs (DL-MSHRs).

3.4.2 Challenges

While DL-MSHR conceptually is a simple but attractive approach, implementing miss handling entries that are flexible and adaptive faces several major challenges. First, unlike the linked list in data structure at the software level, where operations can be easily specified in high-level programming language and executed by a general-purpose processor, here dynamically linking slots needs to be implemented at the hardware level and controlled by a dedicated, low cost logic unit to handle various cases, which is not a straightforward task. Second, since MSHR slots are dynamically formed, additional time may be needed in finding available slots and in locating the position to link the slots. Thus, techniques and optimizations are needed to minimize the delay and power overhead of DL-MSHR, as well as to avoid frequent linking operations. Third, the DL-MSHR array and DAU should be designed in a way that is transparent to other components. In other words, all the original signals to and from the box with dashed boarder in Figure 3.6 should be exactly the same as in the conventional MSHR architecture. This avoids changes and verification efforts to other components, and helps to integrate the proposed scheme in commercial GPUs.

In the rest of this section, we present the detailed design of DL-MSHR, addressing what specific architecture changes are needed to link the slots, how the resources are organized physically and connected logically, what steps are involved in processing primary and secondary misses, when to allocate and free entries and slots, how the DAU is realized using finite state machines, along with four

optimization techniques to further improve the efficiency of DL-MSHRs.

3.4.3 DL-MSHRs

Figure 6 illustrates the conventional explicitly addressed MSHRs and the proposed DL-MSHRs. All the slots in a row share the same block address and a comparator (denoted as "C"), and each slot includes the offset bits of the read/write data, a data buffer, and other related miss tracking information. Figure 3.7a shows a conventional MSHR architecture with 4 entries, each having 4 slots. As a result, if there are more than 4 concurrent primary misses or more than 3 concurrent secondary misses after any primary miss, there will be reservation fails due to MSHR entry-full and merge-full, respectively. However, it is unlikely that every cache line has the same number of outstanding secondary misses. Hence, many slots may still be available even in case of reservation fails.

To utilize the slot resource more efficiently, the proposed scheme decomposes the static entries into a pool of slots. Several slots form a slot *set* as the basic element for dynamic allocation (two slots in the example of Figure 3.7b). Managing resource at the granularity of sets rather than individual slots adds another level of flexibility and helps to reduce slot linking operations as discussed later. A slot set can be dynamically allocated as an independent entry for processing a primary miss, or can be linked after another slot set in an existing entry to increase the capacity of processing secondary misses, forming a "super-entry".

Figure 3.7b shows an example of how 8 slot sets are *physically* organized in

the proposed DL-MSHR architecture, and Figure 3.7c shows one possible *logical* state at runtime. In this logical state, there are 5 super-entries or DL-MSHRs (we use the term super-entry and DL-MSHR interchangeably in this chapter), and the super-entries have varying number of slots. Since there are 8 slot sets with 16 slots in total, maximally this DL-MSHR structure can process up to 8 outstanding primary misses concurrently if all the slot sets are assigned as entries, or handle up to 15 concurrent secondary misses after the primary miss if all the slot sets are linked together as one super-entry. This is significantly more than what conventional MSHRs in Figure 3.7a can handle.

The dynamic allocation is self-adaptive and does not require external interference to dictate when to link or delink. When a super-entry is full and a new secondary miss comes, it is the time to link a free slot set if one is available. When the requested data is returned from lower level and forwarded to the requesters, it is the time to break the corresponding super-entry and free all its slot sets. An internal control unit (i.e., DAU) is still needed to initiate the operations, and several extra bits are needed in each slot set:

Head bit (*H*): this bit indicates whether the slot set is the first set in a super-entry to handle a primary miss.

Linked bit (*L*): this bit indicates if there is another slot set attached to the current one to handle more secondary misses.

Pointer bits (*P*): these bits (e.g., 3 bits in the example of Figure 3.7c) work together with the L bit to specify the ID of the next linked set. This allows the control unit to find the physical location of the attached set.

Set free bit (*S_free*): this bit is set to 1 if all the slots in the current set are unoccupied, so the slot set can be dynamically allocated by the control unit.

Set full bit (*S_full*): this bit is set to 1 if all the slots in the current set of slots are occupied. If another secondary miss comes, a new slot set needs to be linked to the current set. The *S_free* and *S_full* bits are not mutually exclusive, e.g., when a slot set is partially occupied, both *S_free* and *S_full* bits are 0.

Lastly, a counter *nFreeSet* is maintained to track the number of free slot sets in the entire DL-MSHR structure. The counter is simply decremented or incremented whenever the control unit allocates or frees a slot set. Using the counter is a nice solution to avoid ANDing the *S_free* bit of every slot set, which would otherwise be slow. The above extra bits and the counter are all very small (no more than a few bits), which has minimal overhead compared with the slot set.

3.4.4 Operations

With the above architectural changes, we describe the three main operations of DL-MSHRs as follows.

Handling Primary Misses. When a miss is detected by the Tag & Control unit (TCU) in the conventional MSHRs, a search signal is sent to the comparator array to find whether there is a match in the MSHRs. The same signal is now sent to the DL-MSHRs. The block address included in the miss request is compared with the block address in each DL-MSHR. If no match is found (i.e., a primary miss), the *nFreeSet* counter is checked to see if any free slot set is available. If

yes, an allocation signal is passed on to the DAU. A free slot set is selected to serve as a new super-entry that keeps track of the primary miss. The head bit is asserted to indicate the current set is an independent entry. The S_free bit is set to 0 signifying that this entry is currently occupied. The cache block address, offset address and requester ID are recorded in the first slot of the current set, as the slot set may consist of multiple slots. If this primary miss is a write request, the data is written into the data buffer (applicable in the write-back cache). If the above *nFreeSet* counter is 0, it means that the entire DL-MSHR structure has no available slot set to handle any new primary miss. This memory request is stalled until a slot set becomes free, as indicated by *nFreeSet*.

Handling Secondary Misses. When the TCU detects a match in the conventional MSHRs, it generates a merge signal to the matching MSHR. This merge signal is now intercepted by the DAU. The DAU tries to merge the request into the matching DL-MSHR by storing the request in a free slot in the last slot set (i.e., tail slot set) of that super-entry. All the preceding slot sets should have been fully occupied. To locate the tail slot set, the DAU searches from the head slot set and follows the linked bit (L) and pointer bits (P) set by set until reaching the tail slot set, whose L bit should be 0. In the tail slot set, there are 3 possible cases:

(a) At least one slot is free (i.e., *S_full* is 0). In this case, the information of the miss request is recorded in the first available slot in the set. The DAU then modifies the *S_full* bit based on whether the current slot set is full after accepting this secondary miss. For a write miss, the DAU also writes the data into the data

buffer.

(b) No slot is available in the tail set, but *nFreeSet* ¿ 0. In this case, the DAU selects a free slot set to be linked as the new tail set. The old tail set stores the ID of the new tail set in the P bits and asserts the L bit to record the linking information. The new tail set deasserts its head bit, stores the miss request in the first slot (which should be free), and deasserts the the *S_free* bit.

(c) No slot is available in the tail set, and *nFreeSet* is also 0. In this case, no free slot set is available to be dynamically linked in the entire DL-MSHR structure. The miss request has to be stalled until a slot set becomes free, and then goes into the above case (b).

Deallocation of DL-MSHR. A super-entry and all of its slot sets are deallocated and recycled when the requested data is returned from lower memory levels and the data is forwarded back to the requesters. To deallocate, the DAU resets the H, L, P, *S_free* and *S_full* bits of all the slot sets in the super-entry. The *nFreeSet* counter is also incremented by the number of newly freed slot sets. Notice that, although the super-entry is deallocated, the data is still in the cache and subsequent accesses will result in cache hits.

3.4.5 Dynamic Allocation Unit (DAU)

A major task in implementing DL-MSHR is how to design a simple yet comprehensive control unit that can respond to various scenarios correctly and promptly. In this subsection, we present the design details of the *Dynamic Allocation Unit*

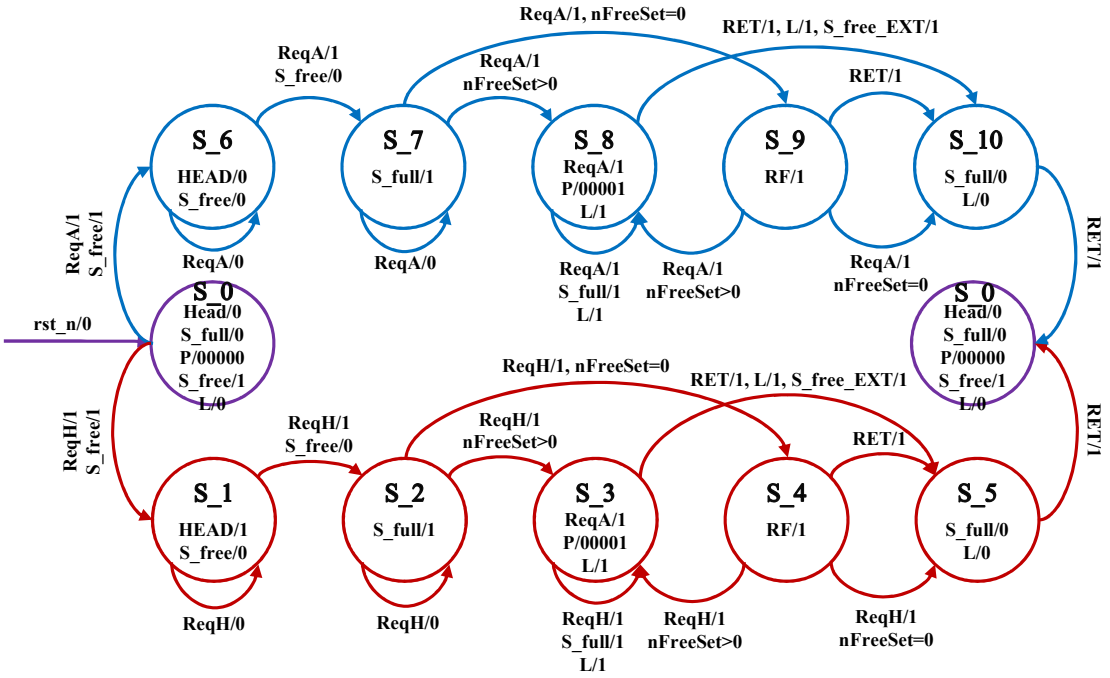


Figure 3.8: The finite state machine used to implement the Dynamic Allocation Unit (assuming 2 slots per set).

(DAU), which serves as an interfacing controller between the original Tag & Control unit and the DL-MSHR arrays. The key component in the DAU is a built-in finite state machine that controls various operations of slot sets. Figure 3.8 shows the finite state machine for a DL-MSHR example with two slots in a set[2]. Following state diagram conventions, the signals on the arrows are inputs and the signals inside the circles are outputs.

There are two main state-transfer paths in Figure 3.8. The lower path is activated by *ReqH* and the upper path is activated by *ReqA*. The *ReqH* is a signal

[2]The minor states for error detection and fault control are omitted in this diagram for clarity, but they are all implemented. Additionally, the same state *S_0* is only replicated in the figure on both sides to avoid long arrows.

that requests a free slot in a head set; if the set is not already a head set, the signal first marks the set as head and then requests a slot. Similarly, the *ReqA* is a signal that requests a free slot in an attached set; if the set is not already an attached set, the signal first transfers the state of the set to "attached".

The lower path containing *S_1* to *S_5* depicts the state-transfer when a free slot set becomes a head set. When *S_0* receives *ReqH* which results from a cache miss, this path is activated and the slot set becomes a head set. Hence, the Head bit is asserted, and the *S_free* bit is deasserted, as shown inside the circle of *S_1*. The *S_1* state implies that the first slot of the current set has recorded the information of a primary miss. At this point, if a new *ReqH* arrives requesting another free slot, the state is transferred to *S_2* and a secondary miss is recorded in the second slot of the current set. With a total of 2 free slots per set, the *S_full* signal should now be asserted. As more secondary misses continue to arrive at the current entry, the DAU checks the *nFreeSet* counter to see if there is any available slot set. If a free slot set is found, the state transfers to *S_3*, and the information of the linked set is recorded. During this state, a *ReqA* signal is sent out that marks the newly found set as "attached" (i.e., activating the upper path for that set, discussed shortly). However, if *nFreeSet* is 0, the state is directly transferred to *S_4* which generates a reservation fail (*RF*) signal. *S_4* may transfer back to *S_3* if a slot set becomes free, as indicated by *nFreeSet* ¿ 0. Later, when the requested data is returned from the lower memory hierarchy, a *RET* signal is generated. This signal is used to release the occupied slots. This includes the transfer from both *S_3* and *S_4* to *S_5*. After the entire set is released, the state goes back to the initial state *S_0*.

Likewise, when *S_0* receives *ReqA* that requests the set to attach to another set, the upper path is activated. The *ReqA*, *RET* and *nFreeSet* are mainly responsible for driving the state transfer. To signify that the current set is used for holding secondary misses, the head bit is set to 0, and the *S_free* is also deasserted to denote an occupied slot, as shown in *S_6*. As more secondary misses arrive, the *S_full* is asserted, leading to *S_7*. Depending on whether a free slot set is available (i.e., if $nFreeSet > 0$), the state transfers to *S_8* or *S_9*. Later when the data is returned, the *RET* signal drives the state to *S_10* and the initial state.

Finally, for deallocation, when the state transfers to *S_0*, a *S_free* signal is asserted which serves as the external free signal *S_free_EXT* to the preceding slot set which is either another "attached" set (from *S_8* to *S_10*) or a "head" set (from *S_3* to *S_5*).

While the prior explanation of how DAU works is detailed, implementing the state diagram in Verilog HDL results in almost negligible hardware overhead of the control logic, as shown in evaluation (Section 6).

3.4.6 Additional Optimizations

Optimization 1: Group slots into sets. The above description of DL-MSHR started with grouped slots without too much explanation. In fact, this optimization has several benefits. First, grouping slots into one set can reduce hardware overhead, as most of the extra bits and resources in DL-MSHR are at the per set granularity. Second, grouping increases the chance of having a free slot

when a secondary miss occurs, thus reducing the frequency of linking another slot set and the associated delay and power consumption. Third, grouping reduces the number of additional comparators needed by DL-MSHR. As each slot set can be used as a separate MSHR entry, the total number of comparators in DL-MSHR is equal to the number of slot sets. For example, in Figure 3.7b, with 2 slots per set, physically DL-MSHR needs 8 comparators. If there are 4 slots per set, the number of comparators would be the same as that of Figure 3.7a. Note that, even in this case, DL-MSHR is still better than Figure 3.7a because the slot sets can be dynamically linked.

Having more slots in a set increases the benefits of the above three aspects, but also reduces adaptivity. Our empirical study shows that having two slots per set offers a much better trade-off than other configurations by a large margin. Hence, two slots per set is used in this chapter as the basic linking unit. In terms of the impact on the critical path, we have used Synopsys design compiler and CACTI 6.5 [68] to evaluate the CAM searching latency of different comparator configurations. A typical 32-entry MSHR design needs 0.2ns to complete the searching of 32 comparators (parallel searching but serial signal driving). When using 64 comparators such as in the 2-slot per set configuration, the searching time only increases slightly to 0.22ns, which is fast enough to match up with the frequency of most commercial GPUs.

Optimization 2: Disable unused comparators. Although physically each slot set has a comparator, the comparator is not used when the set is linked after another one. For example, logically only 5 comparators are active in

Figure 3.7c. Therefore, the unused comparators can be disabled to avoid searching. To realize this, we can reuse the Head bit as a double-function bit. A deasserted Head bit in each slot set indicates that this set is either unused or attached to other set. In both cases, the associated comparator can be safely disabled by using the Head bit as a gated enabling signal. With this optimization, searching through the DL-MSHR arrays still takes roughly the same time (as all the Head bits still need to be searched), but the comparators of unused or attached sets are not activated, thus avoiding the associated power.

Optimization 3: Locate tail set faster. During the operation to link a slot set to an existing super-entry, the DAU needs to locate the tail slot set. If the super-entry contains many slot sets, this may take several cycles. To avoid this delay, an extra pointer that stores the ID of the current tail slot set can be added in the head set in a super-entry. The pointer is updated when a free slot set is linked as the new slot set, and is reset when the super-entry is deallocated. By adding this pointer, the latency to locate the tail set can be reduced to one cycle. We have evaluated this optimization, and simulation results show that the technique does improve performance, although the improvement is not large, around 0.5% IPC increase when averaged over the benchmarks. This is mainly because: 1) super-entries with a large number of linked slot sets are not common, 2) locating the tail set is needed only when linking slot sets, and 3) the latency can be partially hidden by multiple outstanding misses.

Optimization 4: Reserve head sets. We also augment the proposed DL-MSHR with the ability to reserve some head slot sets. In DL-MSHR, it is

Table 3.1: Evaluated benchmarks.

Benchmarks	Abbre.	Ref	Benchmarks	Abbre.	Ref
Backprop	BP	[22]	Transpose	TP	[76]
Bfs	BFS	[22]	Aligned Types	AT	[76]
B+tree	B+T	[22]	AsyncAPI	AA	[76]
Cfd	CFD	[22]	BlackSchole	BS	[76]
Dwt2d	DW	[22]	BinomialOptions	BO	[76]
Heartwall	HW	[22]	ConvolutionSeparable	CS	[76]
Hybridsort	HB	[22]	FastWalshTransform	FWT	[76]
Nw	NW	[22]	Merge Sort	MS	[76]
Srad	SRA	[22]	QR Generator	QG	[76]
StreamCluster	SC	[22]	Radix Sort Thrust	RST	[76]
Cutcp	CUT	[91]	Reduction	RED	[76]
Histo	HIS	[91]	ScalarProd	SP	[76]
Lbm	LBM	[91]	Scan	SCN	[76]
Stencil	STC	[91]	SobolQRNG	SQ	[76]
Sgemm	SG	[91]	Sorting Network	SN	[76]

possible that all the slot sets are linked together as one huge super-entry to satisfy the need of a particular cache line with an unusual number of secondary misses. While this is intended and beneficial in some cases, it is rare that the entire many-core processor has only one primary miss. This can be easily addressed by setting the Head bit of some sets to always be 1 to prevent these sets from being attached to other slot sets. In our design, half of the slot sets are simply reserved to process primary misses, and the other half can be freely linked to other sets. Future work can be done along this interesting line to explore other configurations.

Table 3.2: Simulator configuration.

# of SMs	28
Per-SM limit	48 warps, 8 CTAs
# of Mem partitions	8
L1D cache	16KB, 32-set, 4-way local write-back global write-through 32×8 MSHRs per SM (32 entries, 8 slots/entry)
L2 cache	8×128 KB, 64-set, 16-way allocate-on-miss, write-back 32×4 MSHRs per bank (32 entries, 4 slots/entry)
DRAM	FR-FCFS scheduler GDDR5, 16 banks peak bandwidth 345.6GB/s
SM/L2/DRAM clock	1137/1137/2700 MHz
Warp scheduler	GTO, LRR, Two-Level, SWL

3.5 Evaluation Methodology

We apply the proposed DL-MSHR to both L1D cache and L2 cache and implement in the cycle-accurate simulator GPGPU-Sim 3.2.2 [10]. Key parameters are listed in Table3.2. The L1D and L2 MSHR sizes are typical and in line with existing literature and products. Note that 32 entries are per SM for L1D and per bank for L2, so the GPU has thousands of MSHR entries as a whole. Different MSHR sizes (up to 256×32) are also evaluated to demonstrate the cost-effectiveness of DL-MSHR. The main evaluation assumes GTO warp scheduling policy, and other scheduling policies are also tested. GPUWattch [58] is employed to assess energy consumption.

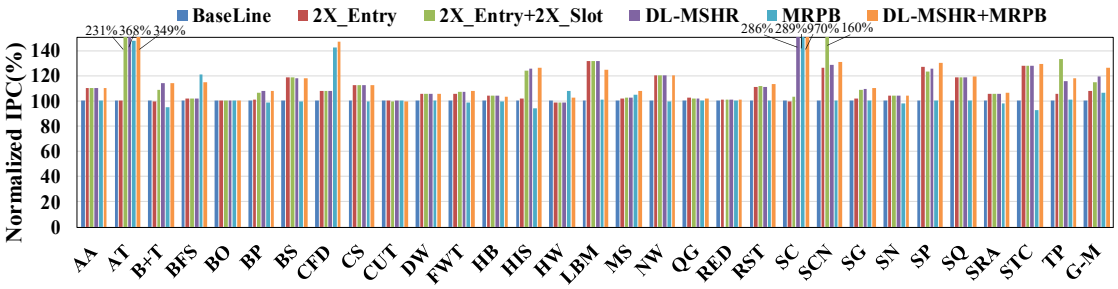


Figure 3.9: Performance comparison over the baseline architecture (normalized to the Baseline).

A wide range of benchmarks from Rodinia [22], Parboil [91], and Nvidia GPU Computing SDK [76] are evaluated that include both compute and memory-intensive ones. Table 3.1 lists the details of the benchmarks. All the benchmarks are run to the end of their execution. It is important to note that memory coalescing in the SMs is already employed, so our evaluation methodology does not artificially increase the number of secondary misses to the cache.

To evaluate hardware cost, we follow previous works (e.g. [47], [61], [94]) to use CACTI [68] to evaluate the "standard" parts of (DL-)MSHR. The data lines are stored in SRAM whereas the CAM structure is stored in flip-flops. All the new components such as the finite-state-machine in DAU and additional bits and control circuits are fully implemented in Verilog HDL and synthesized using Synopsys Design Compiler under NanGate FreePDK 45nm cell library [70] for more accurate area and power evaluation.

We compare the following 6 schemes: (1) **Baseline**: the baseline with conventional MSHRs shown in Table 3.2; (2) **2X_Entry**: doubling the number of MSHR entries of the Baseline (both L1D and L2); (3) **2X_Entry+2X_Slot**: doubling the

number of entries and the number of slots of the baseline, i.e., 4X total slots as the Baseline; (4) **DL-MSHR**: the proposed DL-MSHR with the same total number of slots as the Baseline; (5) **MRPB**: a recent technique that reduces reservation fails by using Memory Request Prioritization Buffers to reorder memory requests in L1D cache and bypassing the cache for selected requests. Note that the prioritization signature used in MRPB is designed specifically for L1D and cannot be applied directly to L2. Also, the MRPB compared here includes both reorder and cache bypassing to improve its performance. (6) **DL-MSHR+MRPB**: applying DL-MSHR on top of MRPB to show that they exploit different opportunities and are complementary.

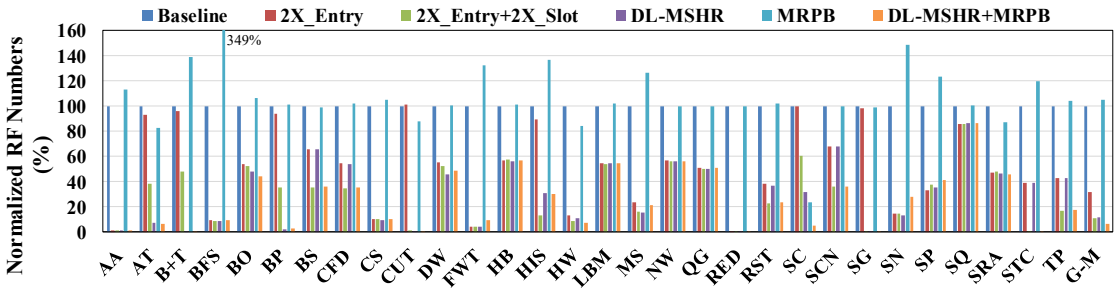


Figure 3.10: Reduction in the number of reservation fails.

3.6 Results and Analysis

3.6.1 Impact on Performance

Figure 3.9 compares the overall IPC improvement of different schemes over the baseline structure. Here the proposed DL-MSHR is applied to both L1D and L2

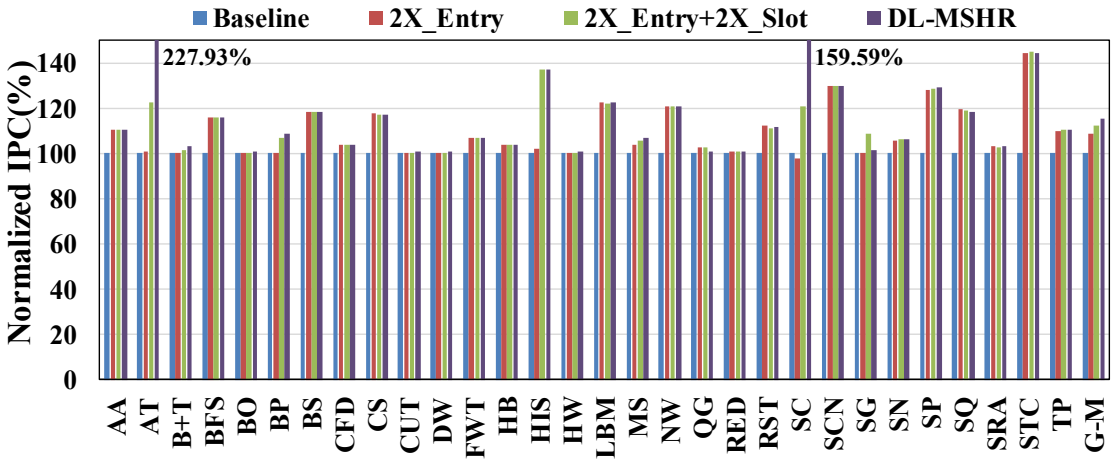


Figure 3.11: Performance comparison with L1D closed.

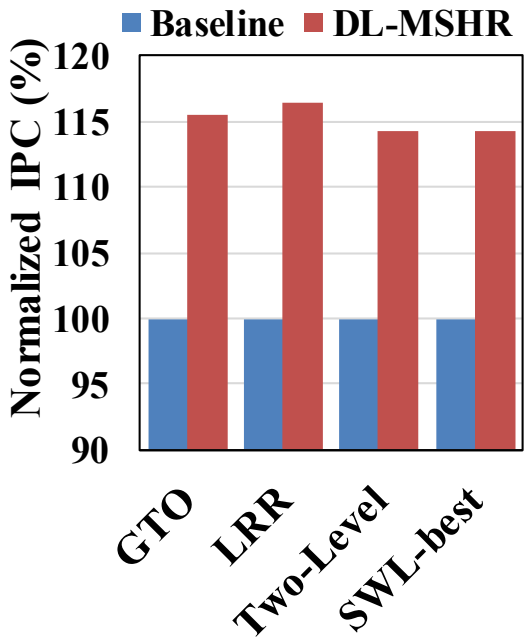


Figure 3.12: Schedulers.

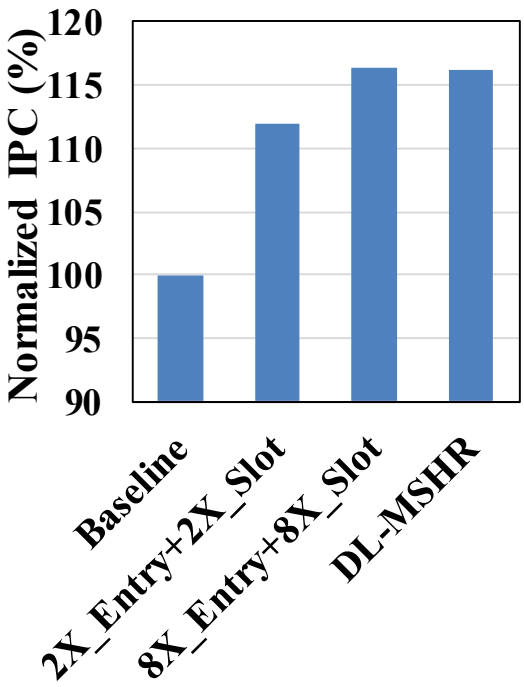


Figure 3.13: MSHR sizes.

cache, and separate results are present in Section 3.6.3. Compared with the Baseline, when the number of entries is doubled, 2X_Entry improves the performance by 8.0% on average. When both entries and slots are doubled, 2X_Entry+2X_Slot improves the average performance by 14.5%. This shows that increasing the number of entries and/or slots can help to relieve some of the pressure on conventional MSHRs. However, some benchmarks such as *CS* and *DW* achieve IPC improvement only when entries are doubled, whereas some benchmarks such as *B+T* and *BP* gain performance only when slots are also doubled. These results are in line with our previous analysis that adding more entries or slots does not work well for all the benchmarks. In contrast, the number of entries and slots in DL-MSHR are dynamically determined based on the cache access patterns of different benchmarks. As a result, the proposed DL-MSHR scheme achieves the best performance among the first four schemes, with an average of 19.2% IPC improvement over the baseline architecture.

For *AT* and *SC*, they both benefit greatly from memory optimizations as *AT*'s kernel mostly consists of memory accesses and *SC*'s access pattern has very low reuse. However, 2X_Entry+2X_Slot does not improve much on SC because the burst secondary misses in SC demand dozens of slots with an entry, which the 2X Slots help marginally. In comparison, DL-MSHR is flexible and can attach up to 64 slots in an entry, thus meeting *SC*'s demand nicely. It is important to note that the above average IPC improvement is calculated based on geometric mean, so the performance improvement is not just because of a few very high bars. For example, among the 30 benchmarks in Figure 3.9, DL-MSHR has around 20% performance

improvement for 8 benchmarks, with over 10% improvement for an additional 9 benchmarks. It is also worth mentioning that the average IPC of DL-MSHR is even 8.0% higher than that of 2X_Entry+2X_Slot which has 4X the number of total slots as DL-MSHR. This highlights the effectiveness and benefits of offering flexible resource allocation in DL-MSHR.

The MRPB compared in the evaluation also improves the geometric mean of IPC by 6.5%, showing that reordering memory requests and selectively bypassing cache help to reduce stalls when MSHRs are heavily used[3]. However, it does not help to balance the uneven slot utilization across different entries, and many resources are still idle even when entry-full or merge-full happens. This issue is addressed by employing the dynamically linked MSHRs. Therefore, the proposed DL-MSHR can be used to complement MRPB, and the resulting DL-MSHR+MRPB improves the IPC by 26.3%, on average, compared with the Baseline.

3.6.2 Reducing Reservation Fails

To provide more insights of the above performance impact, Figure 3.10 compares the number of reservation fails (RFs) normalized to the Baseline (the numbers also include RFs from other sources which accounts for less than 3% in the evaluated benchmarks). On average, doubling the number of entries (2X_Entry) reduces the RFs by 68.3%, and doubling the number of slots on top of this (2X_Entry+2X_Slot) decreases the RFs by 89.2%. In comparison, with the same number of slots as the

[3]The performance gain of MRPB here is different from the original paper as we also used Parboil and NVidia GPU Computing SDK benchmarks.

Baseline, the proposed DL-MSHR can reduce the RFs by 88.1%, and is only slightly less than 2X_Entry+2X_Slot that has 4X the number of slots. It is interesting to see that DL-MSHR has a slightly smaller RF reduction but better IPC improvement than 2X_Entry+2X_Slot. The reason is that, the reduction in RFs is not proportional to increase overall performance, and varies among applications. For example, in the *LBM* benchmark, 2X_Entry+2X_Slot reduces 46.2% of the RFs and achieves 31.5% IPC improvement; whereas in *BP*, the same scheme reduces 64.2% of the RFs, but only increases IPC by 6.3%. These results indicate that the self-adaptive nature of DL-MSHR does not blindly optimize for the overall RF reduction, but rather fine-tunes the number of slots at the granularity of each entry to meet the need of primary and secondary misses at any specific time during execution.

Figure 3.10 also shows that MRPB does not directly reduce the number of RFs, which is expected as MRPB is not designed for that purpose. However, MRPB can help DL-MSHR to further bring down the number of RFs. This is shown in the last bar where RFs are reduced by 93.2%, on average, compared with the Baseline.

The large reduction of RF in DL-MSHR can be mainly attributed to the increase in MSRH utilization. Compared with the conventional MSHR, the proposed DL-MSHR improves MSHR utilization (calculated on a per slot basis) by 53.7% on average. The increase has been observed for every benchmark, although individual results are omitted here due to space limitation.

3.6.3 GPU Architecture Variation

In this subsection, we evaluate the impact of several GPU settings that may be different across GPU generations.

Closed L1 Data Cache. In some recent Maxwell and Pascal based GPUs, the L1D cache are closed (disabled) by default. To evaluate its impact, we disable L1D cache and apply DL-MSHR only to L2 cache. Figure 3.11 compares the performance. MRPB is no longer shown as it works on L1D. 2X_Entry, 2X_Entry+2X_Slot and DL-MSHR improve the Baseline IPC by 8.9%, 12.1% and 15.5% on average, respectively. Comparing Figure 3.9 and Figure 3.11, we can see that the benefits of DL-MSHR may come from both L1D cache and L2 cache, depending on memory access patterns:

(1) when memory requests from the L1D cache of different SMs converge into memory partitions (where L2 caches locate), the DL-MSHR in L2 brings majority of the benefits, e.g., for *AA* and *FWT*, disabling L1D cache only loses 0.2% IPC improvement;

(2) when there are many secondary misses in L1D but the requests for L2 does not exceed the capacity of MSHR in the L2 cache, the benefits mostly come from the DL-MSHR in L1D. For example, *B+T* gets 14.1% performance improvement when L1D is enabled, but the improvement drops to 3.4% when L1D is disabled;

(3) Some applications place pressure on both L1D and L2, and the DL-MSHR in both caches can help, e.g., for *AT*, DL-MSHR improves performance by 228.0% with L2 only, and achieves an additional 140.6% improvement when also applied

to L1D.

Warp Scheduling Policies. In addition to closed L1D, warp scheduling policies may also vary a lot for different GPUs. Figure 3.12 compares the average IPC of benchmarks without DL-MSHR (first bar) and with DL-MSHR (second bar) for GTO, LRR, Two-level [71] and SWL-best [86]. We use SWL with the best static warp limiting numbers (SWL-best) to represent the oracle case for CCWS [86], OAWS [97]. As can be seen from the figure, different schedulers have some but limited impact on the effectiveness of DL-MSHR. In general, these and other schedulers can change the scheduled order and number of warps. This affects data locality and intensity to the cache which, in turn, change the hit and miss numbers. As our proposed scheme enhances miss handling, reduced cache misses may reduce the improvement of DL-MSHR. Nevertheless, cache misses are unavoidable even with the perfect warp scheduler, and warp scheduling does not help much in reducing reservation fails and increasing MSHR utilization. Thus, DL-MSHR consistently achieves sizable improvement under different warp schedulers, from 14.3% in SWL to 16.4% in LRR.

Different MSHR Sizes. While the MSHR sizes of L1D and L2 in our baseline are in line with prior work[47, 61], Figure 3.13 compares the effectiveness of DL-MSHR against other MSHR sizes. We assume 1-cycle access delay to all the conventional MSHR designs regardless of their sizes (thus representing the upper bound of VBF [63] or any other technique that reduces the access delay for large MSHRs); whereas DL-MSHR has 2-cycle access delay due to the access of super-entry and potentially linking of a new set (which is one cycle with the help of

the tail set pointer, although this does not happen on every access). Hence, the comparison is slightly favored towards conventional MSHR designs. As shown in the figure, DL-MSHR with similar resource as Baseline is able to achieve nearly the same performance improvement as 8X_Entry+8X_Slot that has 64X resource of the Baseline, with an average IPC improvement of 16.2% vs. 16.4%. This indicates that the proposed DL-MSHR can be a cost-effective solution to realize very large MSHRs that may otherwise be needed in future GPUs.

Table 3.3: Area and Power of different MHA schemes.

L1D: non-MHA area 0.11mm^2, non-MHA power 42.43mW				
Configuration	MHA Area(mm^2)	MHA Over-head	MHA Power(mW)	MHA Over-head
Baseline	0.00386	3.51%	3.75	8.84%
2X_Entry	0.00739	6.72%	5.67	13.4%
2X_Entry+ 2X_Slot	0.0101	9.18%	7.20	17.0%
8X_Entry+ 8X_Slot	0.120	110%	57.3	135%
DL-MSHR	0.00449	4.08%	4.30	10.1%
MRPB	0.0126	11.5%	15.2	35.9%
L2: non-MHA area 0.97mm^2, non-MHA power 343.28mW				
Baseline	0.0601	6.19%	35.3	10.3%
2X_Entry	0.114	11.7%	63.9	18.6%
2X_Entry+ 2X_Slot	0.216	22.3%	114	33.4%
8X_Entry+ 8X_Slot	3.18	328%	1500	437%
DL-MSHR	0.0611	6.30%	36.1	10.5%

3.6.4 Area and Power Overhead

Table 3.3 summarizes the area and power overhead of different schemes. The results are obtained from Cacti 6.5 and Synopsys Design Compiler. The additional overhead of DL-MSHR over conventional MSHR comes from the extra status bits (head bit, linked bit, pointer bits, set free and full bits), additional comparators and block address fields, a free slot set counter, and the DAU control unit. The overhead of MRPB is mainly from the reorder buffers and related control logics.

To understand the relative impact of hardware cost on the cache subsystem, we put the area and power of the non-MHA part (i.e., the regular tag and data part) of L1D and L2 on top of each table section, whereas the numbers in the main table refer to the MHA part (i.e., MSHRs, comparators, controls, etc.). For instance, the MHA of Baseline in L2 incurs 0.0601mm^2, which is equivalent to 6.19% of the non-MHA part of L2 area. As can be seen, the area and power overhead of directly increasing MSHR sizes quickly becomes substantial, accounting for a significant percentage of regular cache (e.g., 2X_Entry+2X_Slot has 22.3% of L2 area). In comparison, the area and power of the proposed DL-MSHR is very close to the MHA part of Baseline, e.g., within around 0.56% area of Baseline for L1D and within around 0.11% area of Baseline for L2. When taking the previous performance results into consideration, it can be seen that, compared with other optimization schemes, DL-MSHR has higher performance *and* lower area and power overhead.

3.6.5 Impact on Energy

Due to the small hardware overhead, DL-MSHR has minimal impact on the power consumption of GPUs. Therefore, the energy consumption is mainly reduced because of the shorter execution time for reduced static energy. GPUWattch results show that, compared with the Baseline, the proposed DL-MSHR achieves an overall GPU energy savings of 15.7% on average. In comparison, 2X_Entry, 2X_Entry+2X_Slot, and MRPB reduce the energy consumption by 8.7%, 1.43% and 5.1%, respectively. It is also interesting to see that, compared with 2X_Entry, the 2X_Entry+2X_Slot consumes more total energy even though it has shorter execution time. This is because providing additional MSHR resources in 2X_Entry+2X_Slot taxes on the static energy, which illustrates from the energy perspective that naively adding MSHR resources is not an ideal option.

3.7 Conclusion

Contemporary GPUs have an increasing demand for higher memory level parallelism. Consequently, the miss handling architecture must be designed to efficiently track a large number of outstanding memory requests concurrently. In this chapter, we propose a dynamically linked MSHR (DL-MSHR) architecture, which forms MSHR entries dynamically to adapt to application primary and secondary miss behaviors. Evaluation shows significant reduction in reservation fails and large improvement in overall performance, while incurring much less area and power overhead than the alternatives. These results demonstrate the viability and po-

tential benefits of dynamic MSHR structures.

Chapter 4: CAPTURE: Capacity-Aware Prefetch with True Usage Reflected Eviction for GPU Unified Virtual Memory

4.1 Basic Idea

To design a good prefetching policy, two fundamental questions need to be answered: 1) *when to prefetch*; 2) *how many pages to prefetch*. Previous works [108, 87, 60, 33] have explored different prefetchers. Some prefetchers prefetch pages by exploiting spatial localities, while some prefetchers use a tree-structure to enable heuristically prefetching granularity adjustment. Nevertheless, these prefetchers still suffer from low efficiency. When exploiting spatial locality, the fixed prefetching size can lead to low PCIe bus utilization. On the other hand, the complicated structure-based prefetchers have to sacrifice the flexibility of choosing appropriate eviction policies to maintain the correctness of the structure.

To address the above issues, we first propose a capacity-aware prefetcher (CAP) that can adjust prefetching granularity based on the current memory status. Even though the necessity of reducing far-faults prefers prefetching as much as possible, an aggressive data prefetching policy may also cause referenced pages to be heavily displaced, especially under limited memory budgets. This indicates that the prefetching granularity needs to be in accordance with the available memory space. Our proposed prefetcher decreases the prefetching granularity to avoid sat-

urating the memory too early when GPU is running out of memory, and increase the prefetching size when more memory becomes available to reduce page far-faults. Compared to the state-of-the-art work, the proposed prefetcher has superb prefetching flexibility, thus allowing more effective eviction policies to be explored.

Eviction plays a crucial role in UVM, as it enables automatic page evictions to allow programs with memory footprint larger than GPU memory to run directly. However, state-of-the-art schemes [108, 87, 60, 33] still use the classic LRU-based policy that does not accurately reflect the need of residency of pages (e.g., pages that have not been accessed for a long time may potentially get many future accesses). Furthermore, existing eviction schemes share the common performance degradation issue from too small/large granularity and reduced PCIe effective bandwidth, while being complicated by the interplay between prefetch and eviction.

In this paper, we also propose a true usage reflected eviction (TURE) that evicts pages based on their accurate predicted lifetime in the memory. The intuition is that, to avoid page thrashing, we should evict pages that are not reused any more [21]. If the total usage of a page (i.e., page lifetime) can be precisely estimated, we can selectively evict pages that have no further access. This is achieved in TURE by the concept *lease* that specifies the lifetime of the data residing in the memory. We further optimize the lease-based eviction based on the characterization of various GPU memory access behaviors, which substantially reduces the hardware cost. Our final proposed CAPTURE architecture combines the above prefetch and eviction designs in a coordianted fashion. Together, the scheme aims

to "capture" the accurate remaining lifetime of pages, while still being simple, dynamic, and responsive. Evaluation results show that, the prefetcher-only CAP and the combined CAPTURE increase the overall performance by 1.8x and 1.3x on average compared with state-of-the-art, respectively, with minimal hardware overhead. To the best of our knowledge, this is the first work that applies the lease concept to the management of UVM.

In summary, this work makes the following contributions:

- Prefetchers and eviction policies from recent proposals are analyzed, and key issues that negatively impact the GPU UVM performance in these works are identified;
- We propose a novel capacity-aware prefetcher (CAP) to improve UVM performance. By dynamically adjusting the prefetching granularity at run-time, the proposed prefetcher achieves high PCIe bandwidth and significant page fault reduction to improve performance;
- We identify that all pages within the same memory allocation instance present similar access frequencies by profiling different types of benchmarks. Based on this observation, a novel eviction policy *TURE* is developed. *TURE* evicts pages according to the estimated lease in the device memory.

4.2 Background and Motivation

4.2.1 GPU Unified Virtual Memory

A typical GPU program consists of a piece of host code and a piece of kernel code, which runs on the CPU and GPU, respectively. Historically, GPUs follow a "copy-then-execute" programming model [108] because the CPU and the GPU do not share virtual memory space. For the kernel execution, a programmer has to explicitly allocate certain memory space on the GPU and copy data from the CPU to the allocated GPU memory space [25].

To ease the programmers' burden and advance GPU programming paradigm, Nvidia [87, 75] and AMD [2] have started to employ Unified Virtual Memory (UVM) in their newly released products. UVM enables the CPU and the GPU to share the same virtual memory space. This technique completely offloads the memory management overhead from programmers to hardware and OS. In CUDA, when a call to `cudaMallocManaged` is made, only virtual memory is allocated with a pointer returned for further accesses. No physical memory space is immediately allocated on either the host or the GPU. Instead, physical memory is allocated on a first-touch-first-served basis. For example, physical memory is allocated on the host if the CPU first initializes the data (via the pointer). Later when the GPU needs to access the data, a new type of page fault called *far-fault* is triggered, where the GPU memory management unit (GMMU) automatically allocates physical memory on GPU, remaps the virtual memory space to the GPU physical memory, and migrate data over [108, 33]. The entire page fault handling process is transparent

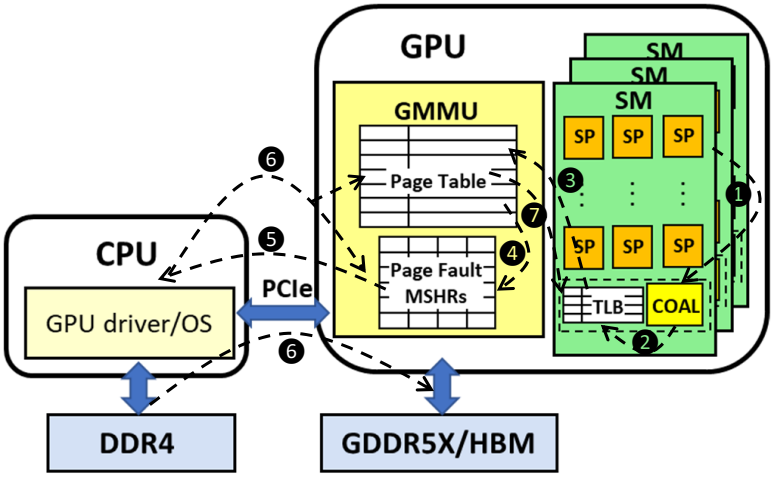


Figure 4.1: Illustration of far page fault handling in CPU-GPU Unified Virtual Memory.

to users.

Figure 4.1 explains how GMMU processes a far-fault. 1 Streaming processors(SPs) generate memory requests upon data accesses. The generated memory accesses within the same warp are first coalesced to reduce redundant memory requests [1]; 2 The Load/Store unit (LSU) in each SM has its own Translation Lookaside Buffer (TLB). A coalesced memory request generated by the memory coalescing unit (COAL) first queries the TLB to see if the TLB can provide a fast address translation (i.e., a TLB hit); 3 Under a TLB miss, TLB forwards the translation request to the GMMU; 4 The GMMU walks through a page table to check whether the address translation entry exists. A far-fault occurs if no address translation has been built for this virtual page (i.e., a page table miss), and an MSHR entry is allocated to record this far-fault; 5 An on-demand page migration request is sent to the GPU driver on the host side through PCIe substrate;

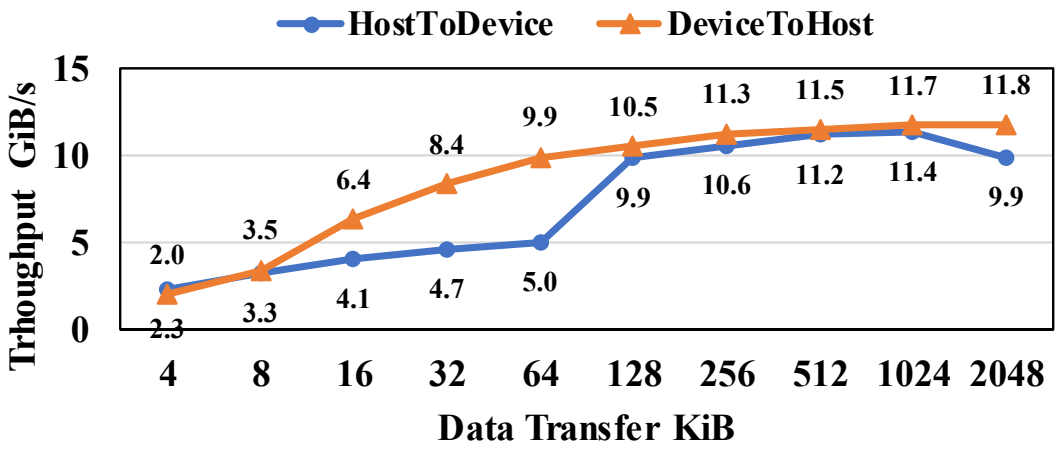


Figure 4.2: PCIe read/write throughput with different transfer sizes. We measure the PCIe effective bandwidth on GTX 1080 Ti, where a PCIe Gen3.0 x16 link is employed to provide 16GB/s link bandwidth.

6 The on-demand page is migrated from the host memory to the GPU memory (a.k.a. paging). Upon the completion of the page migration, the MSHR entry is retired and the page table is updated accordingly by adding a new page table entry to reflect this address mapping; 7 A TLB entry is updated to take this translation information. The GMMU notifies TLB to replay pending requests with the successful address translation.

The introduction of the UVM enables on-demand paging and greatly eases the programming efforts. It allows GPU to be applied in memory intensive workloads even with tight memory budget. However, the long latency brought by far-fault handling and page migrations create new performance and implementation issues. This calls for effective prefetch and eviction strategies for unified virtual memory.

4.2.2 Need for Better Prefetchers in UVM

Prefetching is a promising way to reduce far-fault. Section 4.6 summarizes the related work on UVM prefetcher designs, but even the most recently proposed prefetchers [108, 87, 60, 33] still have significant problems, which are discussed in detail below. These works generally fall into two categories: sequential-local prefetchers and tree-based prefetchers.

4.2.2.1 Issues in Sequential-local Prefetchers

Sequential-local prefetchers (e.g., [108, 60]) prefetch a fixed range of contiguous pages around the faulty page address, thereby exploiting spatial locality. This simple prefetching strategy may not be effective in real applications due to the difficulty in determining the fixed prefetching size. We plot the effective bandwidth of PCIe in Fig. 4.2 by sweeping the data transfer size. The HostToDevice curve corresponds to prefetching. It can be observed that the PCIe throughput is very sensitive to the transferred data size [1]. On the one hand, small prefetching granularity may severely under-utilize PCIe bandwidth. Therefore, frequent page faults and long page migration latency (due to low PCIe throughput) degrade program performance. On the other hand, large prefetching granularity can saturate PCIe bandwidth and, more importantly, exhaust the device memory prematurely, leading to memory over-subscription. Fully occupied memory space triggers ex-

[1] This is mainly because of the constant PCIe protocol overhead and the limited hardware resources (e.g., data buffer size, number of DMA channels, number of outstanding requests, etc.)

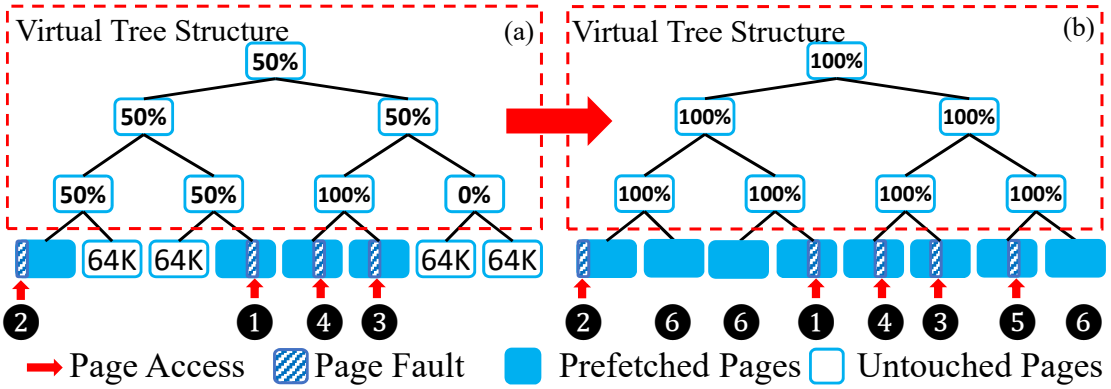


Figure 4.3: The tree-based prefetcher structure that covers 512KiB memory space. Each leaf node refers to a 64KiB block.

cessive page evictions. If not handled carefully, this may cause page thrashing by improperly displacing pages, especially in irregular workloads, and possibly lead to fatal performance degradation [108].

4.2.2.2 Issues in Tree-based Prefetchers

Tree-based prefetchers are used in recent Nvidia GPUs [87] as well as in the latest related work (e.g., [33]). The prefetchers employ tree structures to provide more dynamic prefetching (compared with sequential-local), but incur longer latency and additional limitations in order to maintain correctness.

Figure 4.3 illustrates the heuristic of this kind of prefetchers. A prefetcher maintains a number of binary trees. Each leaf node records the status (i.e., prefetched or untouched) of a 64KiB memory chunk. For example, given a tree with eight leaf nodes, the root of the tree can track the status of a 512KiB contiguous memory space. As shown in Figure 4.3(a), upon a page fault 1, the tree initiates a series of

page prefetches to load the entire 64KiB block onto the GPU (i.e. 4KiB on-demand fetch plus 60KiB prefetch). The corresponding leaf node is marked as prefeteched (in blue). All the intermediate nodes along the path up to the tree root node are also updated by the percentage of their prefetched child leaf nodes. Similarly, 2 3 4 trigger page prefetches within their 64KiB boundary, respectively. As shown in Figure 4.3(b), when another page fault 5 occurs and the corresponding 64KiB block is prefetched, the percentage of prefetched data in the root node now reaches 62.5%, which exceeds a pre-determined 50% threshold. As a result, all the remaining untouched leaf nodes (marked as 6) are scheduled for prefetching. In general, if half of the tree leaf nodes in any branch are already prefetched, the tree predicts that the remaining untouched nodes in that branch will also likely be used and thus prefetches them. Depending on which level of the tree triggers the threshold, the prefetching granularity may vary from 60KiB (a leaf node) to 252KiB (half of tree). In practice, Nvidia GPU driver maintains a binary tree for every 2MiB memory. Therefore, it takes at least 17 (i.e., log(2MiB/64KiB)+1) page faults to trigger migration of the 2MiB memory managed by the tree.

Despite the dynamic prefetching granularity, tree-based prefetchers do not work well as expected in practice for two main reasons. First, upper nodes of the tree may suffer from a long update latency, as the update is performed recursively on all the children nodes. This is also observed in our profiling of GPU benchmarks and may hinder the tree to launch needed prefetches timely. Second, the prefetchers rely on the tree structure to track prefetched pages. Correctness of the tree structure must be maintained during the operations of prefetching and, unfortunately, also

during eviction, as some pages are no longer present. Currently, eviction is done either by evicting the entire tree, or through elaborately designed eviction policy to guarantee that every tree node is tracked properly during eviction processes. Hence, tree-based prefetchers unnecessarily complicate and limit the selection of eviction policies that could otherwise be designed for higher efficiency.

Given the issues in the current sequential-local and tree-based prefetcher designs, a more dynamic, responsive, yet simple prefetcher is much needed.

4.2.3 Problems in Existing UVM Eviction

A main advantage of UVM is that programs with memory footprint larger than GPU's memory capacity (i.e., memory over-subscription) can be run directly [66]. This is accomplished by enabling automatic page evictions from GPU memory when memory is over-subscribed. Unfortunately, some useful pages may also get evicted to the CPU side, leading to negative performance impact. Despite efforts in state-of-the-art works on UVM eviction [108, 87, 60, 33], there still lacks a satisfying eviction strategy that addresses several significant and common problems in existing solutions.

4.2.3.1 Inaccurate Estimation of Page Usage

The most critical problem in existing UVM eviction works is the inaccurate estimation of the usefulness of pages in memory, leading to inferior selection of eviction

candidates. This is largely due to the wide adoption of LRU replacement policy even in the latest works (e.g., [87, 60, 33]). Ideally, the usefulness of a page should reflect the remaining lifetime of the page, i.e., in terms of how many accesses left before the page becomes useless and can be evicted without performance penalty. LRU approximates that by selecting the least recently accessed page, but is still fundamentally different. Pages that have not been accessed for a long time may potentially get many future accesses; it is just that the reuse distance is large. This is particularly evident in Workloads with irregular memory access patterns [60]. To address this problem, we need a new eviction strategy that can directly and accurately estimate the remaining lifetime of pages and then select the least useful page at the time of eviction.

4.2.3.2 Performance Loss due to Eviction Granularity

Another major common problem in existing eviction policies is the performance degradation due to the adopted eviction granularity. For works [108, 60] that evict at per-page granularity, a prefetched continuous memory block may now become fragmented after evictions. As a result, the prefetcher must check the page table for every prefetched address candidate (i.e., no longer once per continuous block) to identify its presence in GPU memory. This checking incurs long delay and can even interfere with demand accesses. Furthermore, maintaining an LRU list at the per-page level also introduces extra delay, particularly considering the large capacity of typical GPU memory [29].

For works that employ the tree structure, as mentioned earlier, Nvidia GPUs [87] evict the whole trees (2MiB) when a page eviction is triggered, to ensure that the memory space of the tree can be freed entirely and recycled by the prefetcher. This granularity of eviction is aggressive and may result in substantial performance loss. Figure 4.4 shows the performance of this aggressive eviction for benchmarks with different memory access patterns, under three cases where memory is over-subscribed (so eviction is needed). Performance degradation rapidly grows to an almost unacceptable point after memory becomes full. In the latest tree-based eviction work [33], this issue is mitigated but at the cost of introducing another source of performance loss. This is discussed next.

4.2.3.3 Reduced PCIe Effective Bandwidth

Reduced effective Bandwidth of PCIe between the CPU and GPU is also a main concern in both per-page and per-tree eviction granularities. For per-page or other small eviction granularities, the DeviceToHost curve in Figure 4.2 shows that the PCIe bandwidth is severely under-utilized, due to a relatively fixed amount of PCIe overhead that would be better amortized over larger transferred data sizes. Because of the slowed transfer, the needed memory space for on-demand pages is not released timely, which eventually leads to stall in the kernel execution.

Meanwhile, inspired by the Nvidia tree-based prefetcher, an eviction policy is proposed [33] that reverses the process of tree-based prefetcher. The eviction policy uses the Nvidia tree structure information as well as hardware resources

to initialize and conduct the eviction process. Although not explicitly explained in that paper, because the tree structures are built and maintained in the driver, the eviction information can only be obtained by frequent communications with the CPU side over PCIe. This put more contention on PCIe, thus reducing the effective bandwidth for the actual prefetch and demand accesses.

4.2.3.4 Challenging Prefetch-Eviction Interplay

Both prefetchers and eviction policies play important roles in the GPU UVM model. When memory is over-subscribed, both of them are active which may raise additional issues. In particularly, it can be challenging to identify whether a specific page is located on the GPU memory or CPU memory. This is needed at the time of prefetching to determine if pages need to be prefetched from CPU. In this aspect, none of the recent works provides a satisfying solution. The eviction policies in Zheng et al. and Li et al.[108, 60] require the prefetcher to check the page table for every prefetching page candidate regardless of the granularity of the prefetcher; the Nvidia eviction policy [87] evicts the whole tree to avoid checking individual addresses; and the eviction policy developed by Gauguly et al. [33] needs to communicate with CPU to get the required information. All of the above impact performance negatively.

In summary, despite the great importance of eviction policies in achieving high performance UVM, current state-of-the-art works suffer from non-ideal selection of eviction candidates, which are further worsened by various performance over-

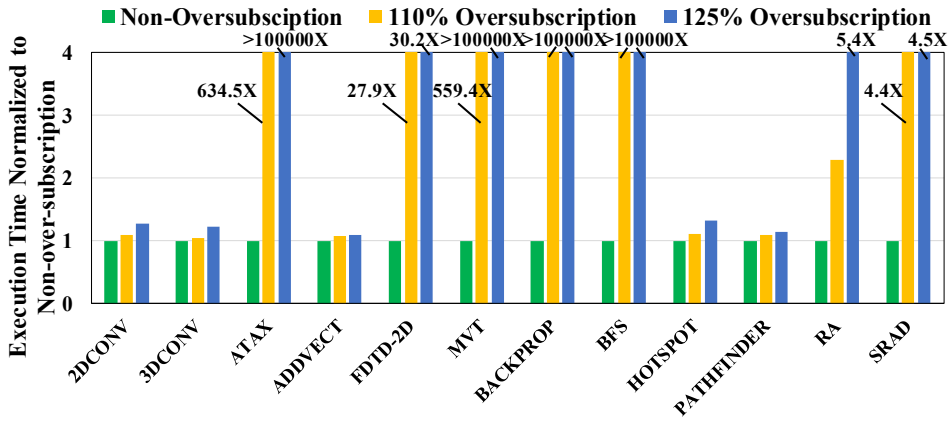


Figure 4.4: Execution time under varying memory over-sub-scription settings. Measurement is done on GTX 1080 Ti.

head due to eviction granularity, PCIe bandwidth utilization, and interplay with prefetch. Therefore, a more effective eviction strategy is much needed.

4.3 Proposed Approach

In this section, we propose a novel, coordinated prefetch-eviction scheme for UVM called *CAPTURE (Capacity-Aware Prefetch with True Usage Reflected Eviction)* which is simple, dynamic, responsive and reflects the true usefulness of pages in eviction. The proposed CAPTURE consists of two main parts as depicted in the blue and red dashed boxes in Figure 4.5 (the gray ones are existing components in GMMU). The *capacity-aware prefetcher (CAP)* is proposed to improve the efficacy of page prefetching. By monitoring the current memory status, the proposed prefetcher dynamically adjusts the prefetching granularity based on the available memory space, which balances the PCIe bandwidth utilization and memory oc-

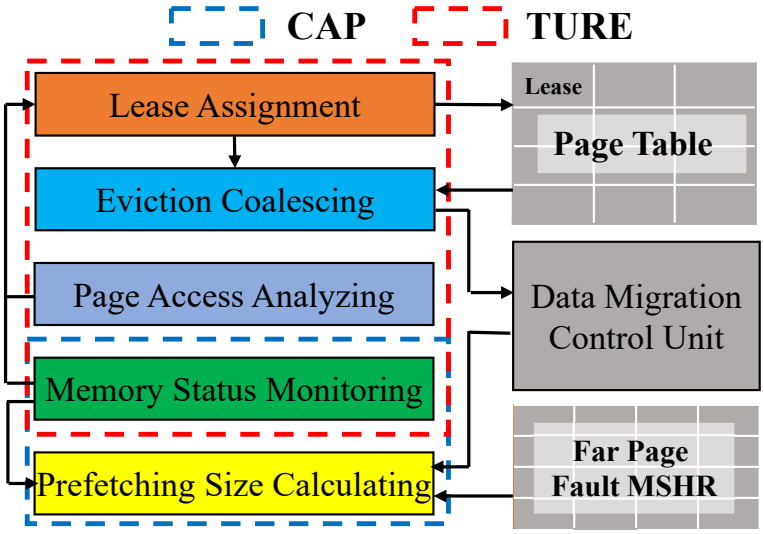


Figure 4.5: Overview of CAPTURE in GMMU.

cupancy. Moreover, a *true usage reflected eviction (TURE)* is developed for page evictions during memory over-subscription. The eviction aims to "capture" the lifetime of the page in the GPU memory by introducing the concept of lease, and selects pages for eviction based on their remaining lifetime to maximize the usefulness. With several additional optimizations, TURE nicely addresses the issues of PCIe bandwidth utilization and performance overhead mentioned in the previous section.

While CAPTURE looks attractive on the high level, the micro-architecture details need to be designed carefully to ensure its effectiveness. For example, the prefetcher needs to dynamically decide the appropriate prefetching size by taking memory status into consideration, while still keeping the hardware simple. Also, unlike lease cache [62], CAPTURE does not involve OS (which would incur

huge storage overhead and intensive computations) and thus does not have the global view of all memory accesses. Additionally, it is challenging to find the most appropriate time for eviction and the evicted data size. Immediately evicting a lease-expired page may cause heavy page thrashing and low PCIe bus utilization due to small eviction granularity. These and other design and implementation details are presented in the rest of this section.

4.3.1 Capacity-Aware Prefetcher

4.3.1.1 Determining Prefetch Granularity

The goal of the proposed CAP is to dynamically adjust the prefetching granularity. The design is based on the following rationale. Large prefetching sizes may help to reduce the number of page faults in the GPU. When there is large available space in the memory, it is beneficial for performance to prefetch large sizes. On the contrary, small available space should correspond to small prefetching sizes to avoid prematurely oversubscribing the limited device memory. In addition, a large allocated memory space usually gets more memory accesses than a small one over the same period of time. Therefore, prefetching size for a given allocated memory space should also be proportional to its size allocated by the function call `cudaMallocManaged`.

Specifically, during the time when a kernel is being launched, CAP assigns a dedicated granularity tracking register for each memory space allocated by `cudaMa-`

`llocManaged`. The register is used to track the prefetching granularity for the on-going page far-faults. When the GMMU begins to service a page far-fault, CAP first inquires the available memory space and then calculates the prefetching granularity threshold based on the eq.4.1,

$$G = M_{avail} \times (M_{req}/(M_{agg} \times c)) \tag{4.1}$$

where G is the calculated prefetching threshold of the current page fault; M_{avail} is the total available memory space; M_{req} is the requesting memory size specified by one of the input parameters of the functional call; M_{agg} is the aggregated size of the memory allocated by all `cudaMallocManaged` function calls; and c is an artificial factor. Without c, when G is equal to the available memory size after calculation, the prefetched pages can saturate the memory directly. To avoid over-subscribing the memory directly, this conservative factor c is introduced to decrease the prefetching threshold. In this paper, we set c to 1.1 empirically, as results are not sensitive to this factor when it is within (1, 2).

Since calculated G may not be a power of 2, we round down G to one of five nearest granularity candidates from 64KiB to 1MiB (e.g. if G is calculated to be 420 KiB, and then the prefetching granularity is 256 KiB). Note that the minimum prefetching granularity is set to 64KiB, which is the same as the size used in the tree-based prefetcher. Also, Figure 4.2 reveals that 1MiB is a sweet point to assure full PCIe bandwidth, 1MiB is set as the maximum prefetching granularity. There is a restriction when calculating prefetching granularity threshold: when the M_{req}

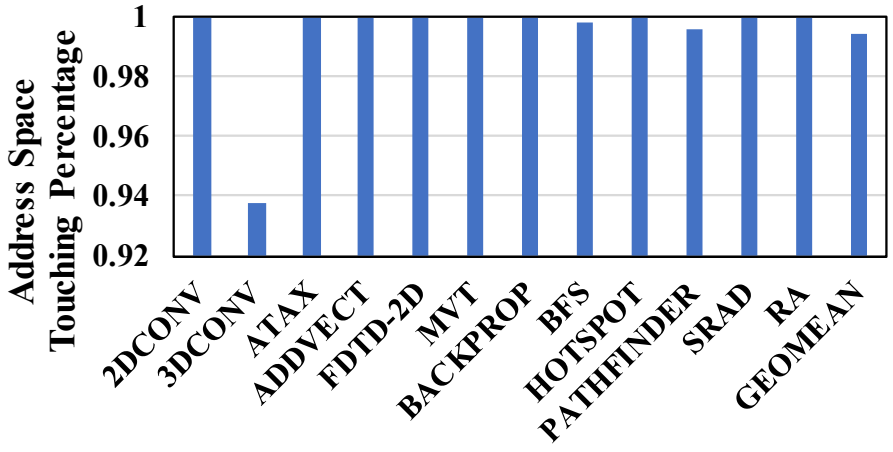


Figure 4.6: Touching percentage of allocated pages.

is smaller than a threshold, the maximum threshold it can select is also rounded down to its nearest prefetching candidate. For instance, if M_{req} is 220KiB, its granularity candidates is from 64KiB to 128KiB.

The reason why our proposed prefetcher works well is that GPU memory access usually reflects certain streaming behavior: the next N pages are likely to be accessed immediately after the occurrence of the current page being accessed [108]. We can apply this on CAP to evaluate if the prefetched pages are eventually accessed in different types of benchmarks. We define *page touch rate* as the percentage of total allocated pages that are accessed during execution. If most of the pages stay untouched at the end, the page touch rate would be low, which indicates prefetching pages may not be accessed. Figure 4.6 shows the page touch rate of different types of benchmarks. From the figure, most of the benchmarks have extremely high page touch rate (i.e., >98%). Only one tested benchmark has relatively low touch rate, but is still above 93%. This indicates that most

of the prefetched pages in CAP are touched (i.e., useful) as expected. Moreover, compared with the tree-based prefetcher shown in Figure 4.3, the simple structure of our prefetcher gives us more flexibility of designing eviction policy.

4.3.1.2 Handling Memory Over-subscription

Based on prior art [108, 33] as well as our results shown in the evaluation section, prefetcher should be enabled under memory oversubscription to reduce page faults and increase PCIe bandwidth utilization. However, if following the calculation of eq.4.1 after memory is fully occupied, CAP would always stay at the smallest prefetching granularity (64 KiB). To address this issue, the proposed prefetcher automatically increases the prefetching granularity to 512KiB when memory over-subscription is detected. The size of 512KiB is selected as the best trade-off prefetching granularity after conducting a series of experiments (presented in Section 4.5.3). This size is also consistent with a prior work [108]. Any invalid 64KiB blocks within this 512KiB address range are scheduled for prefetching.

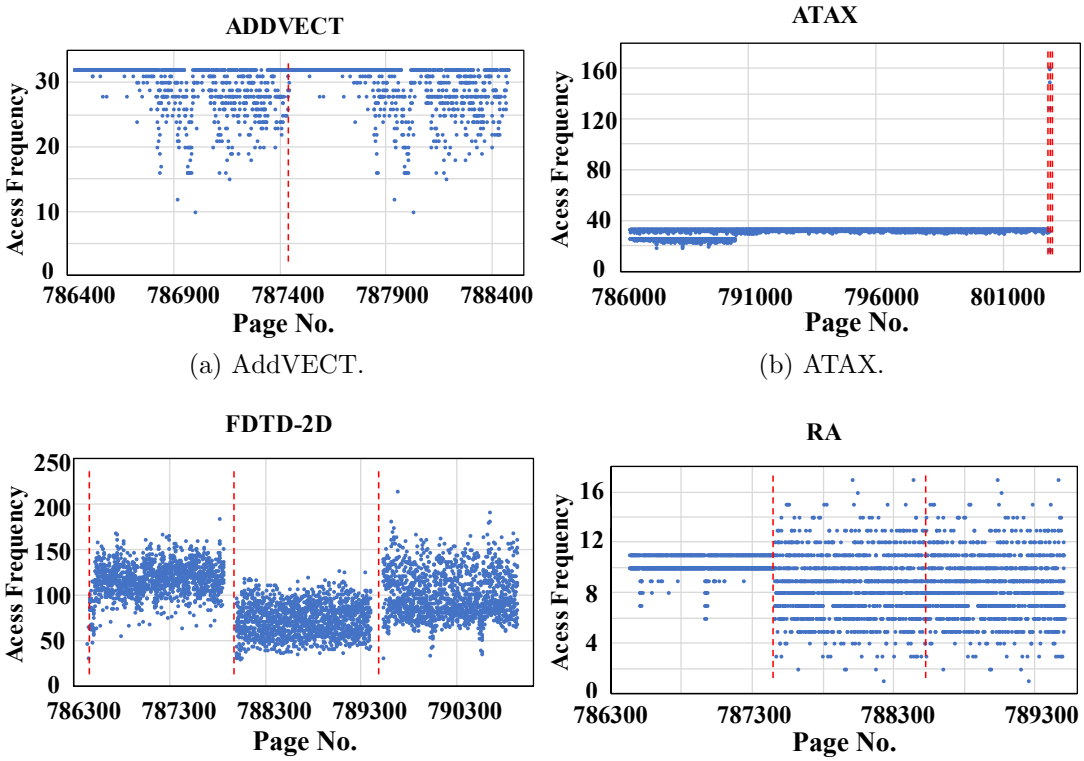
4.3.1.3 Locating Pages during Prefetch

As discussed in Section 2, a key operation of the prefetcher is to identify whether pages are in the device memory or not. In other words, the prefetcher only prefetches pages that are not present in the GPU memory. Previous works either need to perform page table walk for every page or frequently communicate

with the CPU side to get prefetching information. Our proposed prefetcher nicely avoids these problems. As our smallest migration (both prefetch and eviction) granularity is 64KiB, we only need to check whether the first page of a 64 KiB chunk is present in the device memory. Therefore, the worst case of the page table walks is 16 (1MiB / 64KiB = 16), which is much better than the per-page granularity prefetch. In addition, our prefetcher completes this step in the device hardware, thus saving the trouble of communicating with the CPU to frequently synchronize the page residence information. During evaluations, a latency of 45 μs is used for page fault handling, which is quite conservative.

4.3.2 Lease-Based Eviction Policy

To capture the usefulness of pages in memory more accurately, our proposed eviction is based on the concept of *lease*. A lease specifies the lifetime of data in a temporary storage. The concept was initially proposed in distributed file caching [37], and further developed in TLB [9] and cache [62]. Lease is measured in logical time and a lease of p means to keep the data in the storage for the next p accesses. The lease gets updated whenever the associated data is accessed. The data is eventually evicted once its lease expires. When this concept is used in the UVM as the metric to guide evictions, it actually estimates the usage of each page. Even though the concept is straightforward, the implementation can be extremely costly if the number of leases is at scale. For example, a relatively accurate lease is crucial for the scheme to work properly. Therefore, previous work [62] needs a



(a) AddVECT. (b) ATAX.

(c) FDTD-2D. (d) RA.

Figure 4.7: Memory access frequency statistics. The X-axis is the touched page ID across the whole execution period, and the Y-axis is the access frequency of the page. The red dashed lines highlight the different access patterns from `cudaMallocManaged`.

variable per cache line to track the access behavior in the OS. Nonetheless, the GPU memory can have millions of pages (e.g., for a GPU memory of tens of GiB memory capacity and 4KiB page size, which is a common case today). This could incur significant design overhead if each page has a counter to store its lease. Fortunately, we are able to integrate the lease concept with the GPU UVM through several optimizations.

4.3.2.1 Lease Prediction

Our low-cost prediction of page leases is made possible by an interesting observation from our analysis of GPU page access characteristics. Figure 4.7 demonstrates the result of some benchmarks that have *regular* and *irregular* page access patterns as defined in prior art [60]. The X-axis shows the touched page number (page ID) across the whole execution period, and the Y-axis shows the access frequency of the page (i.e., the total number of accesses to the page). The red dashed lines divide the figure into several segments to highlight the different access patterns from memory allocation instances (i.e., `cudaMallocManaged`).

A key observation from the result is that **contiguous pages originated from the same allocation call have similar access frequency**[2]. This observation prompts us that these pages can be clustered into a group and their average access frequency can be used as a lease for all the pages in the group. As the number of memory allocation calls in the kernel is usually several orders of magnitude smaller than the number of allocated pages, it can significantly reduce the design overhead if a lease is given per memory allocation call (rather than per page). Therefore, we define *page group* in this work as the pages allocated by the same function call. For all the pages in a page group, a single lease is used as they have the similar access frequency. We have tested various benchmarks to validate our assumption. It is, however, still possible for some complex workloads to exhibit very fluctuated memory access behaviors. In that case, the workload should require a large cache

[2]While not shown in the figure, all other tested benchmarks support the same observation.

size to cache different data, so these workloads may not be suitable for the GPU acceleration in the first place due to the small size of caches.

According to the analysis of GPU memory access behavior, we propose a *periodical sliding window sampling* technique to periodically sample incoming page requests in the GMMU. This procedure periodically activates the page access analyzing block shown in Figure 4.5. When this block is activated, TURE samples the access frequencies of the incoming page addresses and the corresponding blocking addresses. Contiguous sampling usually requires large storage space to store the sampled information. Therefore, to minimize the storage overhead, it works for D duty cycles in every S cycles (e.g. 1000 duty cycles in every 6000 cycles). During sampling, depending on the page group size, TURE maintains one or more entries for each page group to derive the average access frequency. Having 16 entries turns out to be sufficient in the current design, and increasing the number of entries has minimal overhead due to the small size of an entry (Section 4.5.5. When there is no free entry, TURE ignores the following accesses to a new page group, which should be a rare case. Each entry consists of an access counter and a page diversity tag. The access counter increments by one when there is a reference to the page group. The page diversity tag works as a *set* data structure that only unique pages belonging to the page group is set. At the end of a sampling period, the average access frequency of each page group (F_{avg_access}) is calculated by eq.4.2, where n is the number of entries belonging to the page group, N_i is the number of page accesses from the access counter, and P_i is the number of pages touched by counting 1s in the page diversity tag. The new average access frequency of the

page group is further adjusted in a weighted manner to generate the final lease, as shown in the Procedure.

$$F_{avg\_access} = \sum_{i}^{i \in n} (N_i / P_i)/n \tag{4.2}$$

Although the Procedure approximates and tracks the lease during real time, it is still inevitable to have inaccuracies in the lease estimation, as one lease is used for the whole page group after all. Individual pages have some fluctuation in access frequencies, as illustrated in Figure 4.7. If not dealt with care, it can cause significant page thrashing if the lease is given inaccurately and the pages are evicted immediately after the lease expires. In addition, PCIe bus can be saturated undesirably by these expired pages. Therefore, we further propose to coalesce the evictions caused by lease expiration to mitigate the negative performance impact of inaccurate lease.

4.3.2.2 Eviction Coalescing

As discussed above, the inaccurate lease of a page can result in inefficient page eviction, either too early or too late, and degrades the performance due to page thrashing and memory pollution. Inspired by the warp coalescing concept in the GPU cache, we propose *page eviction coalescing* technique. The size of a coalescing granularity is 64KiB, which is consistent with the minimum prefetching granularity of the tree-base prefetcher. Additionally, this coalescing process makes sure that

the eviction granularity can make use of the PCIe bandwidth. In the proposed design, a coalescing table is employed and each entry is used for a coalescing granule. Once the lease of a page expires, its address information is first sent into the coalescing table. An entry is allocated for the corresponding coalescing granule if it does not exist yet. Then the page offset is set to indicate that the expired page is ready for eviction. In this work, the coalescing table can support up to 256 entries to provide good eviction performance.

Two events can trigger the eviction controller to schedule page eviction. First, it starts to evict pages whenever the coalescing table becomes full (i.e., all entries have been allocated). Under this situation, the least recently updated entry is selected for eviction to release the entry. Secondly, the controller also evicts pages if GPU memory is under over-subscription. In this case, four blocks are selected each time for eviction based on the LRU policy. The larger eviction size is to make enough room for the incoming on-demand pages. Note that the selected pages are evicted no matter if some of them fully expire or not. As a consequence, some useless pages can be evicted for free to help performance.

Predicted smaller lease may result in the early eviction of useful page blocks. To mitigate this effect, we propose a recall mechanism. When a page with expired lease is accessed, the TURE revokes the entire entry, and re-assign those pages with half of the initial lease. As a result, the eviction coalescing brings us two major benefits. Like the effect of a victim cache, the eviction coalescing table temporarily buffers the expired pages to avoid immediate eviction. Also, it consolidates individual expired pages to fully make use of the PCIe bandwidth and reduce write-back

latency.

4.4 Evaluation Methodology

A wide range of benchmarks from Rodinia [22] and Polybench [36] are evaluated that include regular and irregular page access patterns by following the methodology in prior work [60, 33]. Benchmarks *2DCONV*, *3DCONV*, *ADDVECT* and *PATHFINDER* exhibit a fairly streaming page access pattern across all thread blocks' memory accesses. *ATAX* and *MVT* exhibit reuse patterns with a short reuse distance at the kernel start, but then show a sparse page access pattern. *HOTSPOT* shows a periodically burst reuse pattern. *BFS* and *SRAD* have the most page access reuses with a long reuse distance. *FDTD-2D*, *RA* and *SRAD* exhibit large variations on page access frequency for all pages. Similar to prior work (e.g., [60, 33]), due to the impractically long simulation time, we limit the data size of these benchmarks to between 4MiB and 64MiB, with an average of 20.3 MiB. The device memory size is set to 1GiB for non oversubscription experiments, and is shrunk to the corresponding sizes when testing oversubscription cases.

We implemented the CAPTURE in a cycle-based simulator GPGPU-Sim UVM Smart [33], a modified version of GPGPU-Sim [10]. The simulator models the process of page fault handling and data migration. Table 4.1 lists the main parameters we use in this paper.

We compare our approach with four other prefetchers and four other eviction policies. The first configuration is the baseline, denoted by `NP` (short for no

prefetcher), which only migrates a 4KiB page on-demand without any prefetching. A random prefetcher, `RP` randomly evicts one page within the corresponding 2MiB virtual address range of the faulted page. A sequential prefetcher, `SP`, fetches sequentially contiguous 64KiB of pages where the faulted page fall within. The fourth prefetcher is Nvidia's tree-based prefetcher [33], denoted by `TP`. Our prefetcher is denoted as `CAP`.

The first eviction policy, denoted as `RE`, which randomly evicts one page when out of physical memory. The second is a sequential policy, which is a corresponding policy to `SP` and denoted by `SE`. All pages are organized with an LRU list. At the time of an eviction, we first identify the least recently used page in the LRU list and evict a contiguous 64 KiB of pages where the page falls within. Similar to this policy, Nvidia GPU maintains pages into an LRU list in the granularity of 2MiB, that is, an entry is a 2MiB of pages. At the time of an eviction, 2MiB pages are discarded. We call it `NE` (Nvidia Eviction) for short. The state-of-the-art is a tree-based eviction policy, denoted as `TE`, just by reversing the process of Nvidia's tree-based prefetcher by Ganguly et al. [33]. It first finds out a leaf node of 64KiB pages that is the least recent accessed, and then traverses the tree bottom-up to evict any parent node that has more than 50% pages evicted earlier. Our eviction policy is denoted as `TURE`. The no prefetcher case is denoted as NP in the following evaluation.

Table 4.1: GPU simulator configuration.

GPU architecture	Nvidia Pascal
# of SMs	28 SMs @1481 MHz
Per-SM Limit	64 warps, 32 CTA
Page Size	4KiB
Page Fault Handling Latency	45μs
Page Table Walk Latency	100 Core Cycles
PCIe Bus	PCIe 3.0 16x 8GT/s

4.5 Experiment Results

4.5.1 Effectiveness of Prefetching Schemes

Figure 4.8 compares the effectiveness of different prefetchers without memory oversubscription. Results are normalized for clarity, as the run time of different benchmarks varies by orders of magnitude. The result confirms the effectiveness of prefetching as all prefetchers significantly reduce kernel execution time. `CAP` consistently outperforms all other prefetchers, `NP` by 74.6x, `RP` by 52.7x, `SP` by 5.6x, and `TP` by 1.8x, on average. `RP` only has mild improvement over `NP` by 1.4x while `SP` improves the performance by an order of magnitude. Therefore, it indicates that the spatial locality is a critical factor for performance. On top of it, `CAP` further gains 3.1x speedup over `SP` because it takes advantage of large prefetching granularity as the available memory capacity permits. As a result, `CAP` achieves the best performance because of the effective detection of memory status. On the other hand, although `TP` also employs dynamic adjustment on prefetching size, the average granularity of `CAP` is statistically 6$\times$ larger than that of `TP`. Therefore, `CAP` is the winner.

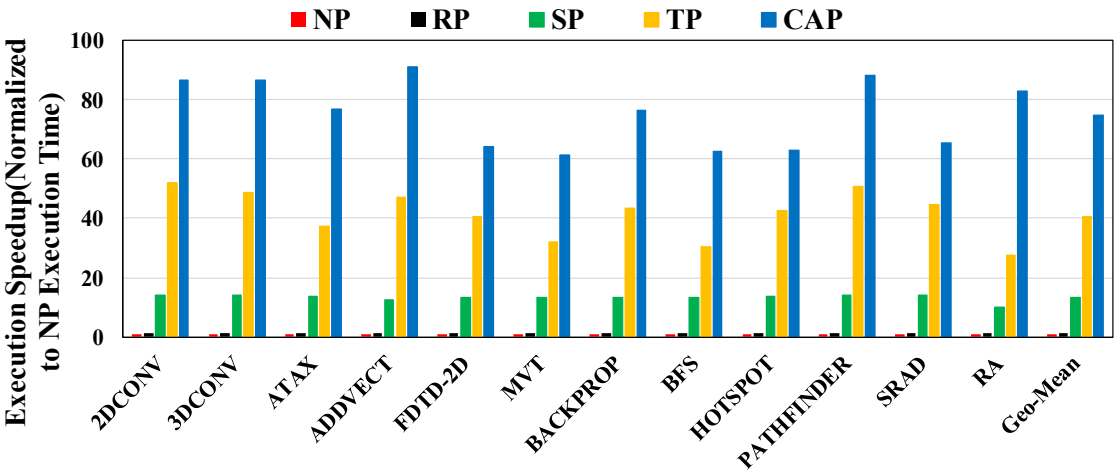
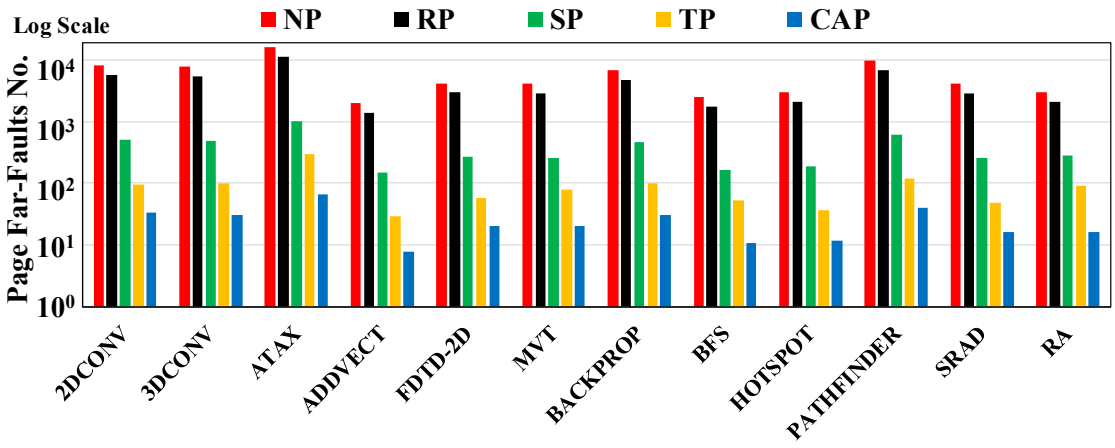
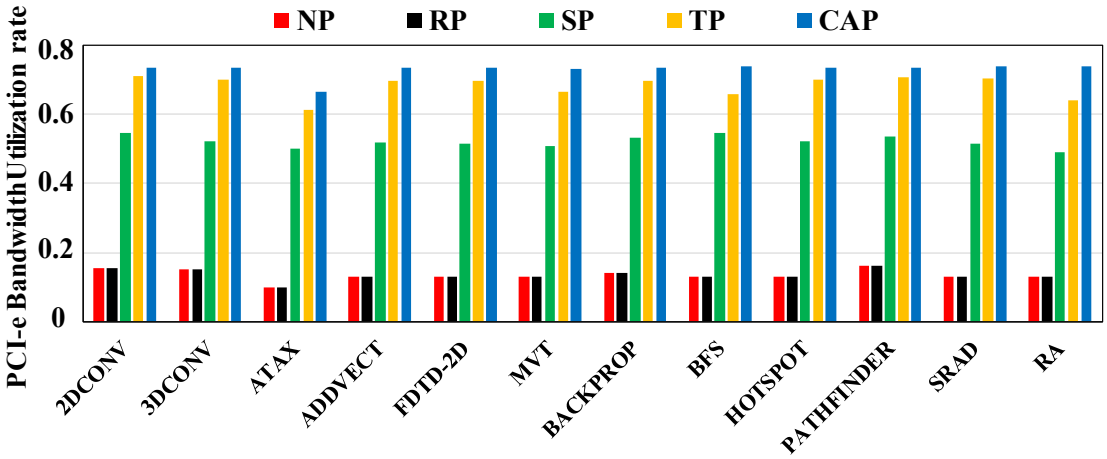


Figure 4.8: Performance speedups of different prefetchers normalized to `NP` without memory over-subscription. The performance refers to as the GPU kernel execution time.

To provide more insights of the above performance impact, Figure 4.9 shows the page-fault numbers and PCIe bandwidth utilization rates with different techniques. With `CAP`, about 99.5% pages are prefetched before being accessed. Hence, we can observe a considerable page faults reduction. In addition, about 99.0% prefetched pages are accessed by the GPU, which indicates the high efficiency of `CAP` prefetcher. On average, `TP` and `CAP` reduces the page-fault numbers by the most, 65x and 234x, respectively. `CAP` is even better than `TP`. Both `TP` and `CAP` can bring considerable page-fault reduction as the two schemes can provide large prefetching granularities, up to 1MiB, though they are realized through different ways. This page fault reduction considerably reduces the time cost by page fault handling process, which further reduces the stall of kernel executions. Profiling results in Figure 4.2 reveal that small data transfer size through PCIe suffers rel-



(a) page fault occurrence numbers with different prefetchers.



(b) PCIe utilization rate during kernel execution with different prefetchers.

Figure 4.9: Benefits comparison of different prefetchers over non-prefetcher configuration.

atively long latency due to constant setup overhead. `NP` and `RP` migrate data with a 4KiB granularity, which not only flood the PCIe bus with massive page migration requests but also keep it working in a very low efficiency. For `CAP`, the large prefetching granularity of 1MiB increases PCIe bandwidth utilization by 5.0x compared with the small 64KiB granularity.

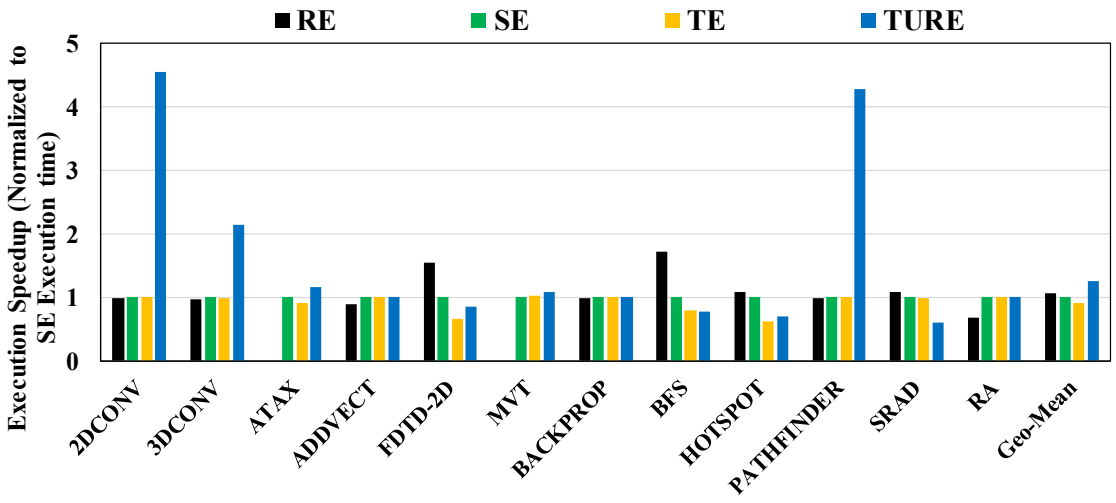


Figure 4.10: Performance comparison with different eviction policies under memory over-subscription.

4.5.2 Effectiveness of Eviction Policies

There are two cases triggering page evictions. One is when GPU memory is run out of; the other is when the coalescing table is full, which enables our eviction policy to become a proactive eviction policy.

To isolate the evaluation of eviction policies, we devise a testing strategy following the work [33]. For all eviction policies, we first run the tree-based prefetcher to load pages into GPU memory and then disable the hardware prefetcher when memory is over-subscribed to avoid any inference to the tested eviction policies. For the generation of memory over-subscription, the data size of all benchmarks is of 110% GPU memory capacity (sensitivity study is shown later). Figure 4.10 compares the performance of all policies for all benchmarks but *ATAX* and *MVT* on `RE`, since they have excessively long running time caused by severe page thrash-

ings. Performance numbers are normalized to SE due to lack of results on RE for the two programs.

TURE gains superior performance on *2DCONV*, *3DCONV*, and *PATHFINDER*, by 4.6x, 2.1x, and 4.3x over RE, on average. The three benchmarks have a clear streaming access pattern, which just confirms that TURE predicts the lifetime of a page precisely. The precise prediction just allows to evict a page right after its use without any delay of the occupancy of memory. It is not surprising that RE has more performance enhancement compared with SE and TE. The essential reason is that SE and TE incurs many page thrashings because they use large eviction granularities without the help of the prefetcher. In contrast, TURE only evicts the expired pages. As long as the lifetime prediction is precise, it does not evict any pages that will be accessed in the future.

TURE is slightly better than the other policies on *ATAX*, *ADDVECT*, *MVT*, *BACKPROP* and *RA*. All these benchmarks access the same pages within a short time and do not reuse them again, which enables TURE to capture and to estimate their accurate average access frequency for each memory allocated space. Benchmarks like RA, BACKPROP having large page access frequency variations for individual page can suffer more page thrashings that counterpart partial benefits.

FDTD-2D, *BFS*, *HOTSPOT* and *SRAD* are observed to have the performance degradation with both TE and our proposed TURE scheme. This results are actually as expected, as their reuse distance is large, which indicates that these benchmarks actually favor small eviction granularity to preserve pages as many as possible for potential reuse. However, with the combination of the prefetcher, this performance

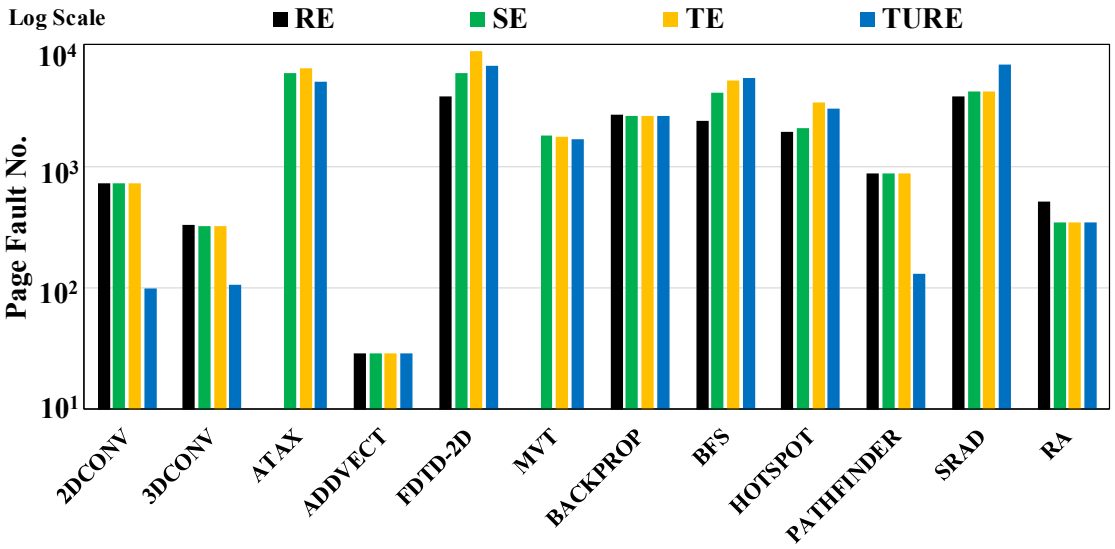


Figure 4.11: Page fault numbers with different eviction policies.

loss can be recovered, as shown in the next subsection.

Figure 4.11 depicts the page fault numbers with different eviction policies. The results shows our scheme incurs fewer page faults, which is correlated with the performance improvement. These results also highlight the importance of prefetchers under memory over-subscription, as page faults are surprisingly high if a page is migrated on-demand under memory over-subscription.

4.5.3 Coordinated Prefetch and Eviction

This section assesses the effectiveness of combined prefetching and eviction. As mentioned before, our proposed prefetcher enforces a fixed prefetching granularity when memory is over-subscribed. To gain insight on how to identify a good trade-off between prefetching granularity and performance improvement, we have

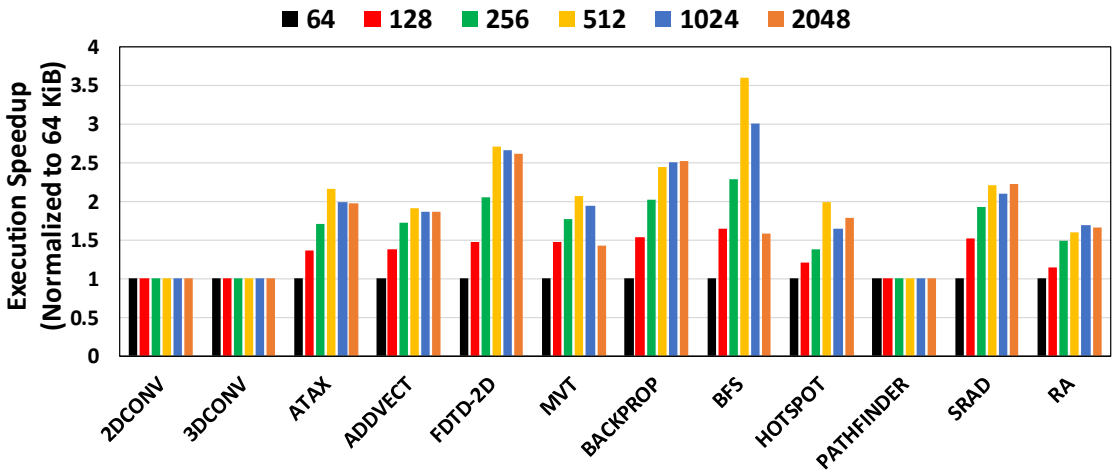


Figure 4.12: Performance comparison of different fixed prefetching sizes (KiB) under memory over-subscription (Normalized to the 64 KiB result).

examined the impact of different prefetching sizes on performance. Figure 4.12 compares the impact on performance improvement by using different fixed prefetching sizes under memory over-subscription. It can be seen that larger prefetching sizes lead to higher performance improvement at the beginning (equal or less than 512 KiB). The reasons behind this improvement are page faults reduction and high PCIe bandwidth utilization. However, when the prefetching size continues to increase after 512 KiB, the performance actually begins to suffer a degradation. When memory is fully occupied, prefetching data excessively can cause severe page displacement by frequently triggering the eviction process. Therefore, based on this study, we select 512 KiB as the fixed prefetching size when memory is over-subscribed in our further evaluation.

In the following evaluations, we name the combination using the concatenation of the abbreviations of the prefetcher and the eviction. For example, `TPSE` denotes

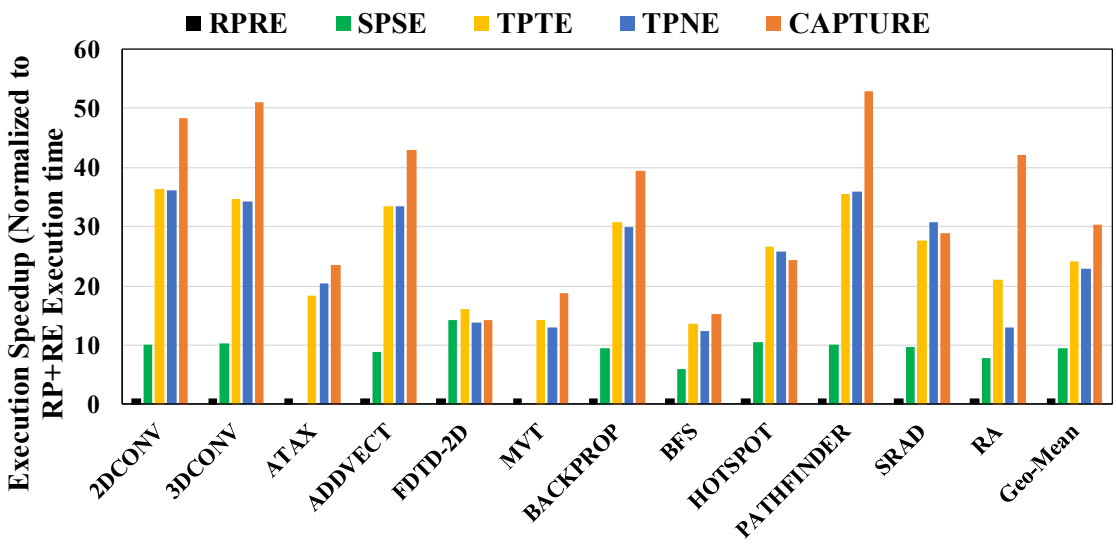


Figure 4.13: Performance comparison of different prefetcher and eviction policy combinations.

using the tree-based prefetcher `TP` and the sequential eviction policy `SE`.

Based on the state-of-the-art works and our proposed scheme, five combined straties are evaluated, `RPRE`, `SPSE`, `TPTE`, `TPNE`, and `CAPTURE`, with respect to the semantics between a prefetcher and an eviction policy. `TPNE` is the current Nvidia's scheme. `TPTE` is the state-of-the-art [33]. For the generation of memory over-subscription, the data size of all benchmarks is of 110% device memory capacity.

Figure 4.13 shows the performance comparison between the five strategies. Results of *ATAX* and *MVT* on `SPSE` are not shown due to the unacceptable long running time. `CAPTURE` consistently outperforms `RPRE`, `SPSE`, `TPTE`, and `TPNE` by 30.4x, 3.3x, 1.3x and 1.3x, respectively, on average. Large granularity prefetching in our scheme can guarantee that PCIe bandwidth are fully exploited throughout the entire execution time. This effectively decreases the page fault numbers and

kernel execution stalls. Our eviction policy tries to evict pages that are less likely to be reused in the near future. It mitigates significant page thrashings under memory oversubscription. The best performing programs are with a streaming access pattern, i.e., *2DCONV*, *3DCONV*, *ADDVECT* and *PATHFINDER*. On *RA*, `CAPTURE` reaches the maximal speedups over `TPTE` by 2.0x and `TPNE` by 3.0x.

Due to some inaccuracy in lease prediction, `CAPTURE` performs a little worse than `TPTE` on *FDTD-2D* and *HOTSPOT*. We explain the reason in detail by taking *FDTD-2D* for example. *FDTD-2D* exhibits phase behaviors of page accesses in Figure 4.7c. The access frequency is very different from phase to phase, but within a phase, the access frequency of a page is close to that of another. We compute the average access frequency in each phase and assign the average as a lease for all the addresses in the phase. However, variations exist from the average to an individual access frequency. These variations incur non-negligible page thrashing for *FDTD-2D*.

It is illustrative to compare `CAPTURE` with `TPNE`, as `TPNE` is the actual practical implementation of Nvidia's memory management policy. The average speedup of `CAPTURE` over `TPNE` is 1.4x. For all programs but *HOTSPOT* and *SRAD*, `CAPTURE` is significantly better than `TPNE`. The reason behind is that the aggressive eviction granularity in NE causes severe thrashing as it could evict highly re-referenced pages, especially for workloads with large reuse distance. `TPNE` is slightly better than `CAPTURE` by 6.4% on *HOTSPOT* and *SRAD*. As analyzed before, the large reuse distance in these two benchmarks reduces the efficiency of our eviction.

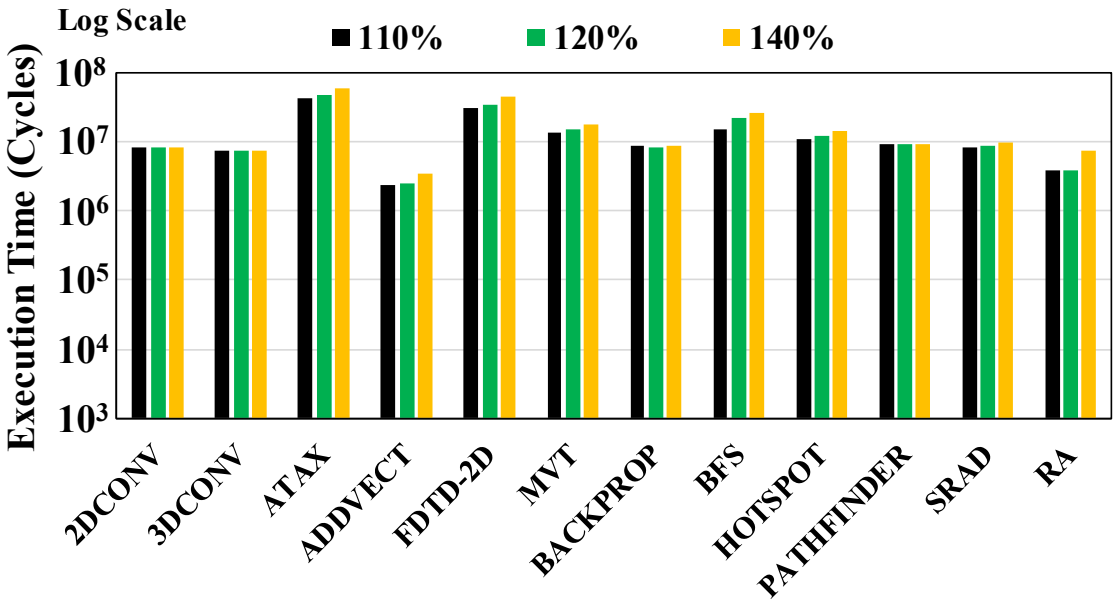


Figure 4.14: Study of memory over-subscription sensitivity.

4.5.4 Memory Pressure Sensitivity

Greater memory pressure usually slows down program performance. To assess the memory pressure sensitivity, we vary device memory capacity to simulate different memory over-subscription scenarios. Figure 4.14 shows the execution time comparison under 110%, 120% and 140% memory over-subscription. Results demonstrate that `CAPTURE` is not sensitive to memory pressure for programs that have a streaming access pattern, e.g., *2DCONV* and *PATHFINDER*. It just indirectly suggests that our capacity-aware prefetching is just a right choice for streaming programs and the lease estimation is very precise. For other programs, `CAPTURE` exhibits a linear, small increase of execution time when memory pressure becomes more, showing a good scalability.

4.5.5 Overhead Analysis

The proposed CAPTURE has low hardware overhead. Despite the various components shown in Figure 4.5, CAPTURE is essentially just a control block that can be generated by a hardware description language, similar to other control blocks such as the network interface. The storage elements in CAPTURE account for the main overhead of the synthesized circuit. Specifically, the overhead mainly comes from three things: (1) A 84-byte buffer, located in Page Access analyzing in Figure 4.5, for periodically sampling incoming requests when analyzing memory access patterns. It consists of 16 entries that are attached by a 10-bit counter and a 16-bit page diversity tag to record different blocks addresses. (2) An overhead of 1.25 MiB, added to the page table. TURE selects eviction candidates based on the remaining lifetime of the pages. Therefore, in the page table, each entry is added with 10 bits to record the remaining lifetime of the corresponding page. Ten bits, or 1024 different values, are more than enough for this purpose based on our profiling experiments. As page table is stored in the DRAM, this overhead becomes trivial when compared with the size of the DRAM. (3) A coalescing table with 256 entries is employed to coalesce eviction candidates It is about 1 KiB in total, as each entry has 32 bits address and a valid bit. Putting together, CAPTURE incurs about 0.01% storage overhead over the DRAM size of Nvidia GTX 1080 Ti.

4.6 Related Work

The most related papers have been discussed in Section 4.2.2, 4.2.3. Other related work is summarized below.

Memory Virtualization in GPUs. Address translation is the crucial part for virtualizing the memory. A number of research has been conducted to reduce the overheads of address translation [11, 14, 15, 16, 17, 18, 80, 77, 7, 90]. Power et al. propose a per-compute unit TLBs and shared page table walker to enable GPU memory virtualization with minimal overheads. Ausavarungnirun et al. [7] proposes a scheme that allows GPU to support multiple page sizes to find the best trade-off between address translation and demand paging latency. Pichai et al [77] develops an optimized MMU for GPUs by proposing modest TLB and page table walker enhancements. Shin et al. [90] explores the impact of orders of servicing page table walks on addresss translation overhead, and proposes a SIMT-aware page table walk scheduler.

UVM in GPUs. Compared with conventional "copy-then-execute" programming models, the UVM in GPUs enables on-demand paging [87, 2, 88]. The driver is responsible for data migration between CPUs and GPUs. Therefore, the UVM eases the programming efforts and enables the GPU to run applications whose dataset is larger than the memory capacity. Nevertheless, this performance impact brought by the new programming model cannot be ignored and has been studied [4, 50, 5, 108, 66, 33, 60]. Agarwal et al. [4] first employs a sequential-local prefetcher to balance the PCIe bandwidth by aggressively prefetching and

throttling page migration based on the bandwidth information. Agarwal et al. [5] further maximizes GPU throughput by exploring page placement strategies based on memory system bandwidth. Kehne et al. [50] and Markthub et al. [66] both enables GPU memory system directly to access system memory to enlarge the device memory capacity. Different from above works only focusing on memory bandwidth or memory capacity issues, The *CAPTURE* provides a comprehensive strategy addressing issues in prefetching and evictions.

The most recent work [53] proposes a thread over-subscription technique that generates more pages faults within a short time to increase page fault handling batch size. It can amortize the GPU runtime fault handling time with increased concurrent page fault handling capacity, and saves eviction waiting time by overlapping the page evictions and CPU-to-GPU page migrations. Nevertheless, this work does not touch any existing prefetching issues, and its per-page granularity eviction policy still encounters low bandwidth utilization problems.

Lease in Systems. Cache leases are initially used in distributed file caching [37], later in most Web caches [31], and recently in TLB [9]. The purpose of their leases is similar, which is to specify the lifetime of data in cache to reduce the cost of maintaining consistency. Li et al. [62] extends lease concept into cache replacement policy. Since the entire replacement operation is completed by the OS and the cache size is relatively small, a complete hash table recording and predicting the lease is built for each data block. However, in the GPU UVM, the entire hardware operation and the large size of the DRAM make it impractical for directly applying lease concept on the GPU UVM. Based on the profiling observation, we made

several key optimizations to greatly reduce the lease overhead on the GPU UVM.

4.7 Discussion

Page sizes. The newly released GPUs begin to support up to 2MiB pages size [7]. Such a large page size is rarely adopted in real applications, because it is difficult for applications to utilize large contiguous regions of memory[7] (and Nvidia does not provide any public materials regarding its large page size application). Despite that, our proposed `CAPTURE` can easily support large page sizes by only revising the prefetching size threshold. This is because `CAP` prefetches pages of continuous addresses without the need of considering page size, and `TURE` is based on the predicted remaining lifetime of pages, which is also independent of the page size. In contrast, the Nvidia tree-based prefetcher and the tree-based eviction policy cannot work with the 2MiB page size , because the whole tree structure represents 512 continuous pages that equals 1GiB when adopting large page sizes. The page prefetching in that case can cause significant memory waste, whereas the eviction can lead to severe page thrashings.

Page Table Walk Latency. Following prior works[108, 60, 33], a 100 core cycles is assumed for the page table walk latency in the above evaluation. We have also tested other page table walk latency, ranging from 50 to 100 and to 150. The overall speedup difference is smaller than 0.01%, thus can be considered as negligible.

Multiple kernels on one GPU. When multiple kernels are running simul-

taneously on a GPU, the kernel executions can potentially affect each other due to different data accesses. It would be interesting to evaluate the effectiveness of UVM under multi-kernel scenarios, but due to the limitation of currently available simulators [33, 7, 108], research along this line will need to be left for future work.

4.8 Conclusion

The UVM programming model is rapidly gaining popularity in GPUs as it removes the limitation of GPU physical memory size and reduces programming efforts. Consequently, an effective virtual memory management strategy must be developed to maintain good performance of UVM in various memory conditions. In this chapter, we propose CAPTURE, a coordinated capacity-aware prefetching and lease-based eviction strategy for GPU UVM. The proposed strategy is able to dynamically adjust prefetching granularity based on the memory status, and evict blocks with less reused opportunities under memory over-subscription. Evaluation results show that our proposed scheme can achieve considerable performance improvement with and without memory oversubscription, while incurring low hardware overhead. These results demonstrate the viability and potential benefits of CAPTURE.

Chapter 5: UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs

5.1 Basic Idea

In this chapter, we aim to enrich the GPU UVM research community by developing a comprehensive UVM benchmark suite consisting of 32 representative benchmarks belonging to different application domains. This suite features unified programming implementation and diverse memory access patterns across benchmarks, allowing researchers to thoroughly evaluate and compare with current state-of-the-art. In addition to traditional benchmarks, the proposed suite also includes more machine learning related workloads, as GPUs have been increasingly used in machine learning tasks. This would help researchers to understand better the role that GPU UVM plays in machine learning acceleration.

The developed benchmarks are evaluated on a Nvidia GTX 1080 Ti GPU with 11GB memory capacity. The code volume is reduced by removing explicit memory management APIs thanks to UVM. Evaluation results show that, if we directly implement/convert benchmarks to the UVM programming model, there is an average of 34.2% slowdown than the non-UVM benchmarks. However, if we augment with proper *manual* optimizations on data prefetching and data reuse, the performance can be restored to almost the same as the non-UVM programming model. This in-

dicates that there is substantial room for UVM research on developing *autonomous* memory management to close the gap between UVM and non-UVM models and possibly exceed the performance of non-UVM. Our experiment also verifies the capability of the UVM-enabled benchmarks to execute successfully under memory oversubscription scenarios, where UVM essentially creates the illusion of a large GPU memory by using a small GPU memory and the CPU memory. While performance degradation is observed compared with a true large GPU memory, this enabling technology opens up new opportunities in accelerating large workloads on GPUs.

The main contributions of this chapter are the following:

- Identifying the need for a benchmark suite for UVM;
- Developing a comprehensive UVM benchmark suite to facilitate the research on UVM;
- Profiling memory access patterns of the benchmark suite, and studying the relevance of the patterns to performance under memory oversubscription;
- Conducting thorough analysis of performance difference between the UVM and non-UVM programming models.

We have discussed the importance of GPU UVM research and the motivation for a benchmark suite in this section. In the remaining of this paper, Section 5.2 describes the proposed benchmark suite in more detail. Section 5.3 explains our evaluation methodology. Section 5.4 presents and analyzes test results. Key

observations drawn from the results and suggestions for future UVM research are highlighted sporadically in that section. Finally, Section 5.5 concludes the paper.

5.2 UVMBench

Benchmarks play an important role in evaluating the effectiveness and generalization when an architecture optimization is proposed. We develop a comprehensive UVM benchmark suite to facilitate the research on the GPU UVM. This suite covers a wide range of application domains marked in Table 5.1. The benchmarks exhibit diverse memory access patterns (more in Section 5.4.1) to help evaluate memory management strategies in GPU UVM. The suite also includes several auxiliary python-based programs to help create and test memory oversubscription cases. The benchmark suite is referred to as *UVMBench*, and has been made available to the GPU research community for both non-UVM and UVM versions (`https://github.com/OSU-STARLAB/UVM_benchmark`). Table 5.1 lists all the benchmarks and their configurations in UVMbench. Table 5.2 compares the UVMbench with some related but limited workloads in several important aspects. The development of the benchmark suite includes the following major efforts.

(1) Re-implement existing benchmarks. We start with combining three existing popular GPU benchmark suites, i.e., Rodinia [22, 23], Parboil [91] and Polybench [78], removing redundant workloads and workload types, and converting into the UVM-based programming model. To implement UVM for these benchmarks, we replace all the host pointers (CPU side) and device pointers (GPU

Table 5.1: List of Benchmarks in the proposed UVMBench.

Application	**Abbr.**	**Domain**	**Kernels**	**Threads Per Block**	**Type**
2D Convolution	2DCONV	Machine Learning	1	256	R
2 Matrix Multiplications	2MM	Linear Algebra	2	256	R
3D Convolution	3DCONV	Machine Learning	1	256	R
3 Matrix Multiplications	3MM	Linear Algebra	3	256	R
Matrix Transpose Vector Multiplication	ATAX	Linear Algebra	2	256	I
Backpropgation	BACKPROP	Machine Learning	2	256	R
Breath First Search	BFS	Graph Theory	6	1024	I
BiCGStab Linear Solver	BICG	Linear Algebra	2	256	I
Bayesian Network	BN	Machine Learning	2	256	R
Convolution Neurak Network	CNN	Machine Learning	6	64	R
Correlation Computation	CORR	Statistics	4	256	I
Covariance Computation	COVAR	Statistics	3	256	I
Discrete Wavelet Transform 2D	DWT2D	Media Compression	2	256	R
2-D Finite Different Time Domain	FDTD-2D	Electrodynamics	3	256	I
Gaussian Elimination	GAUSSIAN	Linear Algebra	2	512/16	I
Matrix-multiply	GEMM	Machine Learning	1	256	I
Scalar, Vector Matrix Multiplication	GESUMMV	Machine Learning	1	256	I
Gram-Schmidt decomposition	GRAMSCHM	Linear Algebra	3	256	I
HotSpot	HOTSPOT	Physics Simulation	1	256	R
HotSpot 3D	HOTSPOT3D	Physics Simulation	1	256	R
Kmeans	KMEANS	Machine Learning	5	1/3	I
K-Nearest Neighbors	KNN	Machine Learning	4	256	R
Logistic Regression	LR	Machine Learning	1	128	R
Matrix Vector Product Transpose	MVT	Linear Algebra	2	256	I
Needleman-Wunsch	NW	Bioinformatics	2	16	I
Particle Filter	PFILTER	Medical Imaging	1	128	R
Pathfinder	PATHFINDER	Grid Traversal	1	256	R
Speckle Reducing Anisotropic Diffusion	SRAD	Image Processing	2	256	R
Stream Cluster	SC	Data Mining	1	512	I
Support Vector Machine	SVM	Machine Learning	2	1024	I
Symmetric rank-2k operations	SYR2K	Linear Algebra	1	256	I
Symmetric rank-k operations	SYRK	Linear Algebra	1	256	R

side) with a unified pointer allocated by the UVM API *cudaMallocManaged*. Also, because the GPU driver is now responsible for data migration, all the explicit memory data migration APIs in each original program need to be removed. This may involve rewriting part of the code around the API calls in some benchmarks to achieve the equivalent functionalities. Moreover, the non-UVM data allocation structure should be adapted to the UVM version. For instance, we have to flatten non-UVM 2D arrays, previously allocated on the host side, into 1D arrays, as no 2D array allocation API is provided in the UVM programming model.

(2) Develop machine learning workloads. As recent machine learning tasks heavily rely on GPUs for acceleration, we also add more machine learning related workloads in our benchmark suite, as briefly described below:

- *Bayesian Network (BN)* is a probabilistic-based graphical model, often used for predicting the likelihood of several possible causes given the occurrence of an event. Our implementation is based on the SJTU version [98] and, during the conversion to UVM, retains the two phases that are accelerated by the GPU: preprocessing where local scores of every possible parent set for each node are calculated, and score calculation where threads obtain the local scores and return the best one.

- *Convolutional Neural Network (CNN)* is most commonly applied to image recognition. It has also been extended to video analysis, natural language processing and many other fields. Our implementation follows the general practice where, for forward propagation, the kernels of convolutional oper-

ations, activation operations and fully connected operations are accelerated on the GPU; and for back propagation, the kernels on error calculations and weight and bias update operations are accelerated on the GPU.

- *Logistic Regression (LR)* is used to predict the probability of the existence of a certain class or event. The cost calculation is accelerated on the GPU. The input of this benchmark is the document-level sentiment polarity annotations which is first introduced in [65].

- *Support Vector Machine (SVM)* is to find support vectors that, collectively, form a hyper plane to separate different classes. In our implementation, the kernel matrix calculation is accelerated on the GPU. The code is based on the Julia project [81] and converted to UVM.

Listings 5.1 and 5.2 show the partial code of the sigma update function in the SVM benchmark, which demonstrates the re-implementation process and newly added benchmarks. Several unrelated variables are omitted for simplicity. Listing 5.1 is the code without UVM, while Listing 5.2 is the code with UVM during runtime. As the traditional programming model requires explicit memory management, the program in Listing 5.1 has to allocate memory space on the device by calling *CudaMalloc* (lines 12-20). It also needs to call *CudaMemcpy* APIs (lines 22-24 and lines 28-31) before and after the kernel launch to explicitly migrate the required data between the host and the device. In contrast, the UVM programming model in Listing 5.2 unifies the memory space of the host and the device. By calling *cudaMallocManaged* APIs (lines 6-7), the code allocates bytes of managed

memory. The allocated variables can be accessed by the host and the device directly, and are managed by the Unified Memory system of the GPU. In Listing 5.2, when this Sigma_update function is called in the main function (line 1), the variables, defined by *cudaMallocManaged*, are passed into the function, and the device kennels can directly access these variables. Therefore, device variable definitions and memory management APIs are removed (i.e., lines 6-7 in Listing 5.2 vs. lines 12-20 & 22-24 & 28-31 in Listing 5.1). It can be seen that the UVM programming model greatly reduces the code complexity.

```
1  Sigma_update(int *iters, float *alpha, float *sigma,float *K, int *y, int
       l, int C)
2  {
3    //Define variables on the device
4    float *dev_alpha = 0;
5    float *dev_sigma = 0;
6    float *dev_K = 0;
7    int *dev_y = 0;
8    int *dev_block_done = 0;
9    float *dev_delta = 0;
10   void *args[10] = {&dev_iters, &dev_alpha, &dev_sigma, &dev_K, &dev_y,
         &dev_block_done, &grid_dimension, &dev_delta, &l, &C};
11
12   //Allocate memory space on the device memory
13   cudaMalloc(&dev_iters, sizeof(int));
```

```
14  cudaMalloc(&dev_alpha, l*sizeof(float));
15  cudaMalloc(&dev_sigma, l*sizeof(float));
16  cudaMalloc(&dev_K, l*l*sizeof(float));
17  cudaMalloc(&dev_y, l*sizeof(int));
18  cudaMalloc(&dev_block_done,
19      grid_dimension*sizeof(int));
20  cudaMalloc(&dev_delta, 1*sizeof(float));
21
22  //Data migration: Host to Device
23  cudaMemcpy(dev_K, K, l*l*sizeof(float), cudaMemcpyHostToDevice);
24  cudaMemcpy(dev_y, y, l*sizeof(int), cudaMemcpyHostToDevice);
25
26  /*Kernel Launch*/
27
28  //Data migration: Device to Host
29  cudaMemcpy(iters, dev_iters, sizeof(int), cudaMemcpyDeviceToHost);
30  cudaMemcpy(alpha, dev_alpha, l* sizeof(float),
        cudaMemcpyDeviceToHost);
31  cudaMemcpy(sigma, dev_sigma, l* sizeof(float),
        cudaMemcpyDeviceToHost);
32
33  //Free allocated  memory space
34  cudaFree(dev_block_done);
```

```
35    cudaFree(dev_delta);
36    cudaFree(dev_y);
37    cudaFree(dev_K);
38    cudaFree(dev_sigma);
39    cudaFree(dev_alpha);
40    cudaFree(dev_iters);
41  }
```

Listing 5.1: *Sigma_Update* function in SVM with non-UVM.

```
1  Sigma_update(int *iters, float *alpha, float *sigma, float *K, int *y, int
       l, int C)
2  {
3    int *dev_block_done = 0;
4    float *dev_delta = 0;
5    void *args[10] = {&iters, &alpha, &sigma, &K, &y, &dev_block_done, &
         grid_dimension, &dev_delta, &l, &C};
6    cudaMallocManaged(&dev_block_done, grid_dimension*sizeof(int));
7    cudaMallocManaged(&dev_delta, 1*sizeof(float));
8
9    /*Kernel Launch*/
10
11   cudaFree(dev_block_done);
12   cudaFree(dev_delta);
```

```
13 }
```

Listing 5.2: *Sigma_Update* function in SVM with UVM.

(3) Optimize data prefetch. In our experiment, we observe that directly converting to the UVM programing model from the non-UVM model can lead to performance degradation, as UVM has to track memory accesses and migrate data to destinations. Therefore, we add an optimization, namely asynchronous prefetching, before each kernel launch by calling the provided API *cudaMemPrefetchAsync*. The purpose of this optimization is to exemplify that hardware prefetchers may bring considerable performance improvement in UVM, as shown later in evaluation results. Users of our benchmark suite can easily enable or disable this optimization by changing the macro definition in the Makefile.

Listing 5.3 shows the code in the Backprop benchmark after enabling the above asynchronous prefetching. The program uses CUDA streams to manage concurrency in GPU applications. Different streams can execute their corresponding commands concurrently. To prepare asynchronous prefetching, it first creates different streams (lines 2-11). With different streams, the prefetching APIs (lines 13-16 and 23-24) prefetch the required data asynchronously. As the data have been fetched in the device before the kernel is launched, the Unified Memory system does not need to stall the kernel and handle page faults. Therefore, the data migration overhead in the UVM is mitigated under asynchronous prefetching.

```
1 //Create streams for asynchronous prefetch
2 cudaStream_t stream1;
```

```
3  cudaStream_t stream2;
4  cudaStream_t stream3;
5  cudaStream_t stream4;
6  cudaStream_t stream5;
7  cudaStreamCreate(&stream1);
8  cudaStreamCreate(&stream2);
9  cudaStreamCreate(&stream3);
10 cudaStreamCreate(&stream4);
11 cudaStreamCreate(&stream5);
12
13 cudaMemPrefetchAsync(input_cuda, (in + 1)*sizeof(float), 0, stream1);
14 cudaMemPrefetchAsync(output_hidden_cuda, (hid + 1)*sizeof(float), 0,
       stream2);
15 cudaMemPrefetchAsync(input_hidden_cuda, (in + 1)*(hid + 1)*sizeof(float
       ), 0, stream3);
16 cudaMemPrefetchAsync(hidden_partial_sum, num_blocks*WIDTH*sizeof(
       float), 0, stream4);
17
18 //Performing GPU computation
19
20 bpnn_layerforward_CUDA<<<grid, threads, 0, stream5>>>(input_cuda,
       output_hidden_cuda, input_hidden_cuda, hidden_partial_sum, in, hid);
21 cudaDeviceSynchronize();
```

```
22
23  cudaMemPrefetchAsync(input_prev_weights_cuda, (in + 1)*(hid + 1) sizeof
        (float), 0, stream1);
24  cudaMemPrefetchAsync(hidden_delta_cuda, (hid + 1)*sizeof(float), 0,
        stream2);
25
26  bpnn_adjust_weights_cuda<<<grid, threads, 0, stream5>>>(
        hidden_delta_cuda, hid, input_cuda, in, input_hidden_cuda,
        input_prev_weights_cuda);
27  cudaDeviceSynchronize();
```

Listing 5.3: Enable Prefetching in *Backprop* with UVM.

(4) Optimize data reuse. Data reuse can also mitigate performance overhead of UVM. This is because if the useful data resides in the device memory for longer time, fewer page faults may occur. To investigate the impact of data reuse where multiple (same) kernels access the same data during the runtime, we add the option to run multiple iterations of a kernel execution to create this type of data reuse opportunities (i.e., the same kernel reuses the same data in different iterations). Users can change the number of iterations (≥ 1) by modifying the macro in each benchmark program file.

Benchmarks in the proposed UVMBench are all implemented in CUDA and can be run on Nvidia GPUs. This suite includes both the non-UVM version (original) and the UVM version implementation for performance comparison. There are no

algorithmic changes when developing the UVM version of the benchmarks. This ensures fair comparison between the traditional programming model and the UVM programming model. Consequently, the observed performance changes are mostly attributed to the difference between programming models rather the algorithms.

Some previous works [33, 24] and the Nvidia SDK present a limited number of UVM-enabled workloads to demonstrate the effectiveness of the UVM or their proposed ideas. Table 5.2 compares the existing benchmarks with our proposed UVMbench in five important aspects. Compared with the existing benchmarks, UVMbench presents more workloads from different domains. In particular, UVMbench includes machine learning workloads to explore the possibility of applying UVM techniques in data-intensive machine learning applications. Moreover, UVMbench provides diverse memory access patterns and supports memory oversubscription.

5.3 Evaluation Methodology

Our evaluation methodology is designed to enable a set of experiments that test the proposed benchmark suite. To investigate the impact of memory access behaviors on UVM, we need to profile memory access patterns of each benchmark. Direct performance comparison is also needed between the UVM and non-UVM implementations. As the driver is responsible for data migration under UVM, the impact on PCIe bandwidth should also be examined. Additional experiment is needed to evaluate the UVM performance under memory oversubscription scenarios.

Table 5.2: UVMBench vs. other benchmarks or benchmark suites.

Benchmarks / Benchmark Suite	**# of Workloads**	**Test in Real Hardware**	**Machine Learning Workloads**	**Diverse Memory Access Patterns**	**Oversubscription Support**
Workloads in [33]	14	✗	✗	✗	✓
Workloads in [24]	6	✓	✗	✗	✓
Nvidia SDK [3]	1	✓	✗	✗	✗
UVMBench	**32**	✓	✓	✓	✓

Table 5.3: Evaluation Platform Setup.

CPU	Intel Xeon E5-2630 V4 10 Cores 2.2 GHz
Memory	DDR4 16GB x 4
PCIe	PCIe Gen3x16 16GB/s
Operating System	Ubuntu 18.04 64bit
GPU	Nvidia GTX1080Ti
Driver version	440.33.01
CUDA	CUDA 10.2
Profiling Tools	nvprof, Nvidia Visual Profiler, NVBit

To conduct the above experiments, we employ an Nvidia GTX 1080 Ti GPU with the Pascal architecture. We use the Nvidia Binary Instrumentation Tool (NVBit) [96] to extract the global memory access patterns of the UVMBench suite. NVBit provides a fast, dynamic and portable binary instrumentation framework that allows users to inspect/instrument instructions. we use two Nvidia official profiling tools to profile the performance related data of benchmarks: nvprof, a command line tool to collect and view profiling data, and Nvidia Visual Profiler, a GUI to visualize the application performance. Table 5.3 includes more details of the CPU-GPU platform.

5.4 Results and Analysis

5.4.1 Memory Access Pattern Profiling

To study the relationship between memory behaviors and UVM efficiency, we first profile memory access patterns of each benchmark. In this experiment, NVBit is

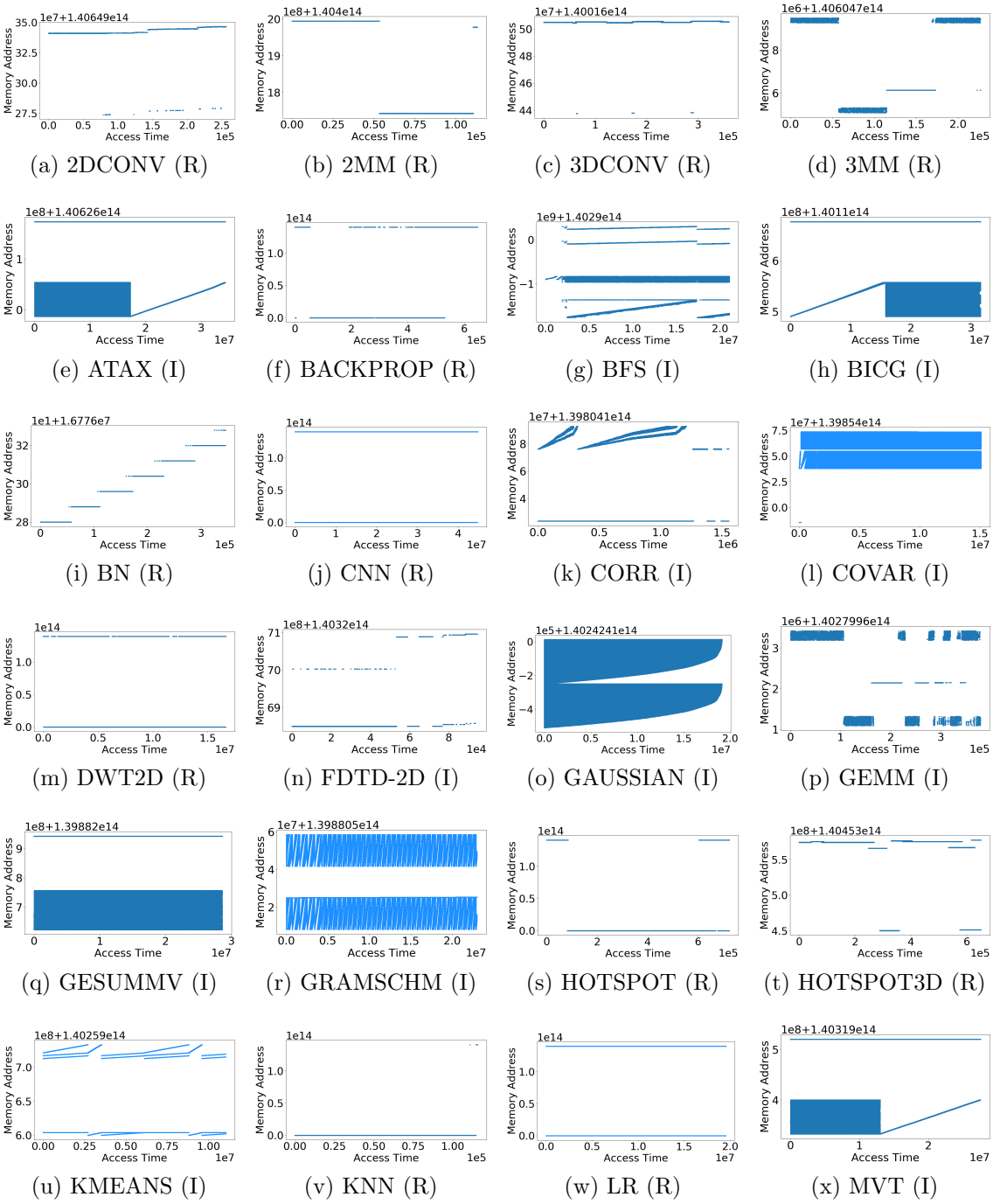
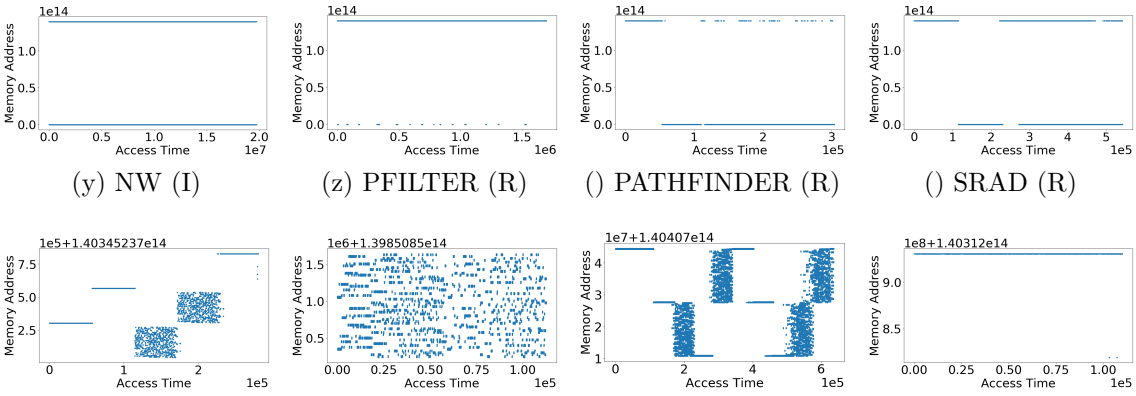


Figure 5.1: Memory access patterns of benchmarks in UVMBench.



(y) NW (I) (z) PFILTER (R) () PATHFINDER (R) () SRAD (R)

() SC (I) () SVM (I) () SYR2K (I) () SYR2K (R)

Figure 5.1: Memory access patterns of benchmarks in UVMBench (continued).

used to generate memory reference traces by injecting the instrumentation function before performing each global load/store. The memory traces are plotted in Figure 5.1. The horizontal axis corresponds to the logical access time, and the vertical axis shows the accessed memory addresses.

As can be seen from the figure, benchmarks in the UVMBench suite exhibit diverse memory access patterns. They can be generally classified into *regular* and *irregular* memory access patterns, as indicated after each benchmark name as (R) or (I) in Figure 5.1 (and as indicated in the "Type" column in Table 5.1). This classification follows the same classification method as [60]: if benchmarks access only a small number of memory pages at any point of time, they are classified as *regular* benchmarks; in contrast, benchmarks with large unique memory pages access at a given time are identified as *irregular* benchmarks. For regular benchmarks (e.g., 2DCONV, 2MM and so on), they exhibit a streaming access pattern. These benchmarks access only a small number of memory addresses and seldom exhibit

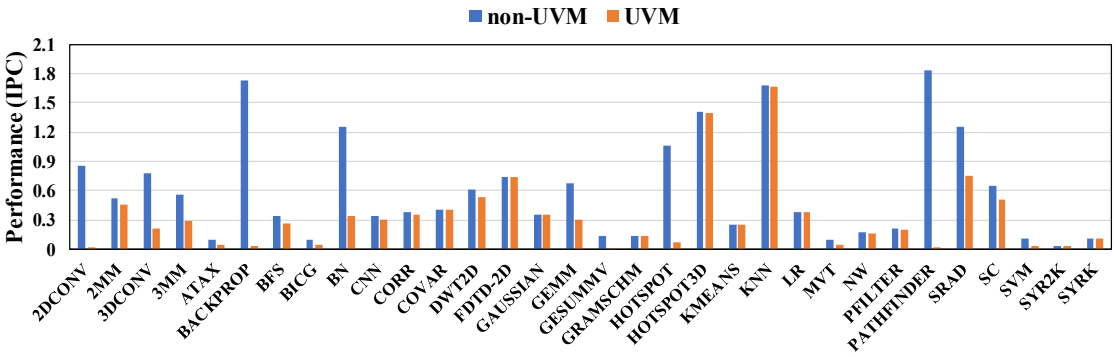


Figure 5.2: Direct UVM conversion in UVMBench leads to large performance degradation vs. non-UVM.

data reuse within the kernel. In contrast, irregular benchmarks show very different memory access patterns: accessing many memory addresses at a given time (e.g., ATAX, BICG, GAUSSIAN), repeatedly accessing the same memory address over time (e.g., COVAR, GRAMSCHM), or accessing random addresses (e.g., SC, SVM). Note that benchmark *NW* is classified as irregular, as it exhibits a sparse, localized and repeated memory accesses, although this is not quite visible in the figure due to the scale. In the experiment of memory oversubscription presented later in Section 5.4.4, we find that benchmark performance is highly related to memory access patterns.

5.4.2 UVM *vs.* non-UVM Performance

a. Performance of Direct UVM Conversion

As mentioned earlier, while UVM greatly eases programming efforts by removing explicit memory management, this is achieved at the cost of certain perfor-

mance overhead, particularly with naive/direct conversion to UVM. Figure 5.2 compares the performance of all the benchmarks in the non-UVM and UVM programming models. The IPCs are obtained from Nvidia *nvprof*. Across the benchmarks, the performance of the UVM version has an average of 34.2% slowdown compared with the non-UVM one. These results are expected as the page fault handling causes large performance overhead for kernel execution. Under the UVM programming model, data is allowed to reside in other location (e.g., on the CPU side) while a kernel is executing. When the required data does not reside in the GPU DRAM (page fault occurrence), the kernel has to be stalled while waiting for the data to be fetched from the CPU side. In the non-UVM version, programmers have made sure that data is always available on the GPU side.

Among these benchmarks, we can observe that *2DCONV*, *BACKPROP*, *HOTSPOT*, *GESUMMV* and *PATHFINDER* have the most significant performance drop in the UVM implementation. The reason is that, for these 5 benchmarks, the data migration time accounts for majority of the entire execution (over 80%), and their kernels have little to no data reuse and are only invoked once. A considerable amount of stall time occurs during the one-time execution of the kernels to wait for data, and the fetched data is not used again. These factors lead to the observed large performance degradation. However, as shown shortly, the performance degradation can be greatly mitigated with some additional programming efforts.

b. Restoring UVM Performance via Data Reuse

Data reuse can mitigate UVM performance degradation by reducing the occurrence of page faults. As mentioned earlier, we study the impact of data reuse by

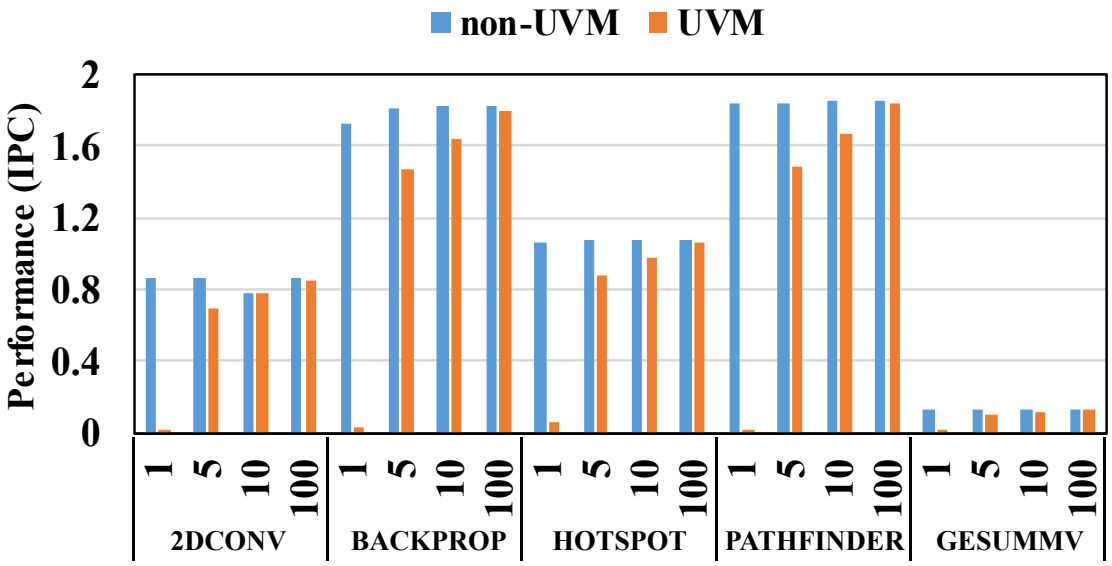


Figure 5.3: Performance of UVM restores with increased number of kernel invocations.

modifying the number of times a kernel is invoked. Figure 5.3 plots the change in performance as we increase the kernel invocation times (there is no kernel execution dependency between consecutively invoked kernels). It can be seen that the performance of these benchmarks under UVM is rapidly improving with more invocation and eventually approaches to the performance of non-UVM. Except for the first executed kernel, the following kernels in the GPU program may reuse the data that has been fetched during the execution of the first kernel, and fewer page faults would occur. The results confirm that more data reuse leads to smaller data migration overhead.

Observation/Suggestion: Although data reuse is artificially introduced in the software program in this experiment, it prompts us that if applications exhibit

significant data reuse opportunities, either inherent or created through architecture optimizations, UVM can be an attractive model that provides flexibility while having little performance overhead.

c. Restoring UVM Performance via Data Prefetch

Nvidia provides a runtime API *cudaMemPrefetchAsync* that enables asynchronous data prefetching. Through this API, data can be prefetched to the device memory before the data is accessed by a kernel on the GPU. This reduces the occurrence of page faults. To study the impact of prefetching on UVM kernel execution performance, we augment all the benchmarks in UVMBench with such prefetching capability. Figure 5.4 shows the results from the above 5 benchmarks that experience the largest performance drop in UVM.

It can be observed that the performance of these benchmarks improves considerably after this optimization and is close to the performance of the non-UVM version. The geometric mean of the slowdown has decreased from 95.8% to merely 0.7%. The improvement comes from the fact that kernel execution is now rarely stalled as data has already been fetched in the device memory before being accessed. While not shown, the performance of other 27 UVM-version of the benchmarks also restores to very close to the non-UVM version after using asynchronous prefetching.

Observation/Suggestion: Besides data resue, another alternative to restore performance degradation of UVm is data prefetching by employing the runtime API *cudaAsyncPrefetch*. In theory, page faults can be completely eliminated if there is an oracle prefetecher that is able to load any required data into the GPU

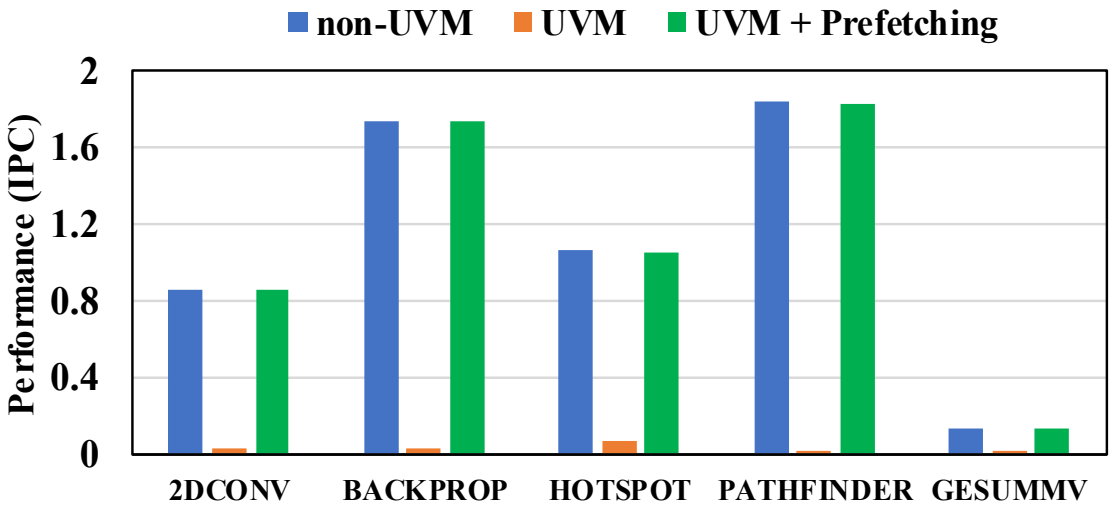


Figure 5.4: Performance of UVM restores by enabling prefetching.

memory before the data is accessed. That can serve as an upper-bound of future UVM prefetech schemes.

It is important to note that, we achieve data reuse and data prefetch in the above experiments by manually modifying the software programs. In other words, these optimizations are realized on the software side and requires additional programming efforts. This is not the intention of UVM that aims to reduce programming efforts. In practice, what is needed is innovation in architecture research that can achieve similar level of data reuse and prefetch but is transparent to programmers. Facilitating research along this line is what our UVMBench suite is created for.

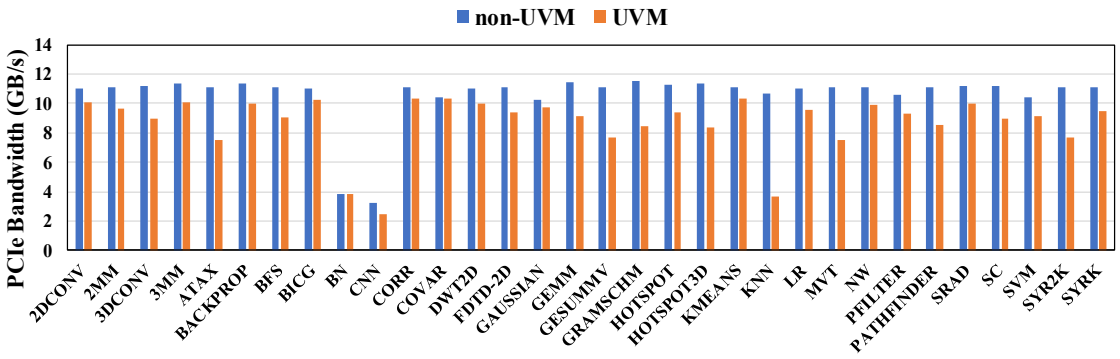


Figure 5.5: Achieved PCIe bandwidth of non-UVM vs. UVM during data migration.

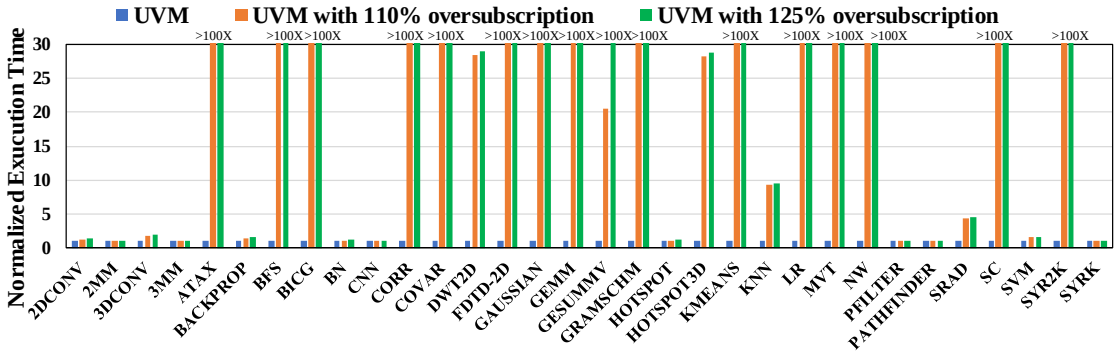


Figure 5.6: Change in benchmark execution time when GPU memory oversubscripted (normalized to no memory oversubscription).

5.4.3 Effect of Data Migration on PCIe Bandwidth

The performance of data migration between CPU and GPU also closely relates to the effective PCIe bandwidth. Under the UMV programming model, variable sized on-demand data is transferred from the CPU memory to the device memory. To understand performance trade-offs, it is worth studying the effect of UVM data migration on the PCIe link. Figure 5.5 compares the achieved PCIe bandwidth with non-UVM and UVM programming models during data migration. On aver-

age, the achieved PCIe bandwidth of UVM is 15.2% lower than that of non-UVM. In general, the larger the transferred data size is, the higher the effective PCIe bandwidth can achieve. This is mainly because of the constant PCIe protocol overhead and limited hardware resources (e.g., data buffer size, number of DMA channels, number of outstanding requests, etc.), so the overhead can be amortized better with larger transferred data. Since the non-UVM model copies the entire allocated data chunk to the GPU memory before execution, this results in relatively high effective bandwidth. In contrast, the migrated data size in UVM is usually much smaller than the non-UVM one as only on-demand data is migrated through the PCIe bus (usually smaller than 1MB). Note that benchmarks *BN* and *CNN* in UVM and non-UVM both exhibit low effective PCIe bandwidth, because the sizes of allocated variables in these two benchmarks are all small (less than 4KB), and even the entire chunk of allocated variable transmission cannot fully utilize the PCIe bandwidth.

Figure 5.5 also shows that, among UVM benchmarks, the effective PCIe bandwidth may vary a lot. The variation is mainly caused by the hardware prefetcher inside the GPU. For example, Nvidia has implemented a tree-based hardware prefetcher in their GPUs, which heuristically adjusts the prefetching granularity based on access locality. The difference in memory access patterns across benchmarks put the hardware prefetcher in different degrees of efficacy. More detailed discussion on UVM hardware prefetchers can be found in other papers such as [33, 105, 60].

Observation/Suggestion: The above results on the effective PCIe bandwidth

indicate that hardware prefetchers that are currently employed in GPUs cannot fully utilize PCIe bandwidth. Thus, future research is much needed to continue developing and optimizing GPU hardware prefetchers that are UVM-aware.

5.4.4 Oversubscription

A major advantage of UVM is to enable kernel execution when memory is oversubscribed. Performance under memory oversubscription can be significantly reduced since part of the data now needs to be brought from the CPU memory. Despite this, UVM is still very attractive, as such memory oversubscription is not possible under non-UVM. To quantify the performance degradation when the GPU memory is oversubscribed, we run all the benchmarks in the suite under various memory capacities. As different benchmarks have different required memory footprint, to create memory oversubscription, we modify the available memory space through the *cudaMalloc* runtime API. The required memory footprint is set to be 110% and 125% of the available memory space in the GPU physical memory. Figure 5.6 shows the results. As expected, all the benchmarks suffer considerable performance degradation under memory oversubscription. The more memory is oversubscribed, the more performance degrades.

From Figure 5.6, we also observe that many of the benchmarks can complete execution with 2-3x slowdown under memory oversubscription, whereas other benchmarks suffer from a significant performance penalty or even crash, marked as >100X in the figure (e.g., LR uses the cublas library which cannot support mem-

ory oversubscription and leads to crash). For the former, we find that the main performance overhead is caused by kernel stalls when waiting for the eviction of pages to create space for newly fetched data. These benchmarks usually have a streaming access pattern (Section 5.4.1). With this pattern and the LRU eviction policy in Nvidia GPUs, the evicted data does not affect kernel execution as the evicted data is not reused any more. Therefore, the performance overhead mainly comes from the waiting time of page eviction. For the latter, the large performance penalty mainly comes from severe page thrashings, which repeatedly migrate the page back and forth between the GPU and the CPU. This usually occurs when a benchmark has a short data reuse distance so the evicted data is needed/reused within a short time. Note that, although the degradation seems large, UVM is still much better non-UVM which does not allow kernels to run at all if the memory is oversubscribed.

Observation/Suggestion: The significant performance degradation under memory oversubscription suggests that the current eviction policies are doing a poor job at selecting the best candidate pages to evict, thus causing severe page thrashings and limiting the amount of memory that can be oversubscribed. This may be possibly because existing eviction policies are not designed specifically with supporting UVM in mind. We urge researchers to develop more effective eviction policies that can select evicted data more accurately or even proactively to make space for expected data accesses.

5.5 Conclusion

The Unified Virtual Memory (UVM) programming model has been introduced recently in GPUs to ease the programming efforts and to allow kernel execution under memory oversubscription. This chapter identifies the need for representative benchmarks for GPU UVM, and proposes a comprehensive benchmark suite to help researchers understand and study various aspects of GPU UVM. Several observations and suggestions have been drawn from the evaluation results to guide the much needed future research on UVM.

Chapter 6: Conclusions and Future Work

In the past decade, GPU has become the most import computing platform for accelerating general purpose computing tasks. As the memory subsystem of the GPU was originally designed for streaming image processing, it has become the bottleneck to prevent performance improvement when running irregular workloads. Therefore, it is necessary to optimize the GPU memory subsystem to make it suitable for current general-purpose workloads. In this work, several memory optimizations focusing on different parts are proposed to improve GPU performance on irregular workloads.

6.1 Summary

Chapter 1 provides the background of GPUs. It also discusses current memory inefficiencies of different parts of memory in modern GPUs, and the potential opportunities to improve the performance. We also introduce current mainstream GPU simulators and emphasize the necessity of developing new benchmarks to facilitate GPU memory research.

Chapter 2 investigates the largely unexplored opportunity of L2 cache access reordering. And we introduces *Cache Access Reordering Tree (CART)*, a novel architecture that can improve memory subsystem efficiency by actively reordering

memory accesses at L2 cache to be cache-friendly and DRAM-friendly.

Chapter 3 identifies the inefficiency of current Miss Handling Architecture in modern GPUs. Based on this observation, we propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that dynamically forms MSHR entries from a pool of available slots. This approach can self-adapt to primary-miss-predominant applications by forming more entries with fewer slots, and self-adapt to secondary-miss-predominant applications by having fewer entries but more slots per entry.

Chapter 4 targets the new techniques in GPUs, called Unified Virtual Memory (UVM). Our experiment and analysis reveal that even state-of-the-art schemes still incur significant GPU kernel performance degradation due to the inefficient page fault handling mechanism in the current UVM programming model. In this chapter, we propose *CAPTURE (Capacity-Aware Prefetch with True Usage Reflected Eviction)*, a novel microarchitecture scheme that implements coordinated prefetch-eviction for GPU UVM management. *CAPTURE* utilizes GPU memory status and memory access history to dynamically adjust the prefetching and "capture" accurate remaining page reusing opportunities for improved eviction.

Chapter 5 tries to enrich the GPU research community with a new benchmark suite. As UVM is attracting significant attention from the research community to develop innovative solutions to these problems, in this chapter, we propose a comprehensive UVM benchmark suite named *UVMBench* to facilitate future research on this important topic. The proposed *UVMBench* consists of 32 representative benchmarks from a wide range of application domains. The suite also features unified programming implementation and diverse memory access patterns across

benchmarks, thus allowing thorough evaluation and comparison with current state-of-the-art.

6.2 Future Work

GPUs will still be the mainstream accelerators for general purpose computing workloads in the near future. In this section, we would like to discuss some research topics that we would like to study in the future.

- *Intelligent data prefetching on traditional GPU programming model.* Current GPUs have adopted High Bandwidth Memory (HBM) as their device memory to improve the performance of data-intensive workloads like Deep learning applications, as HBM can provide much higher bandwidth than the traditional memory. The current GPU memory hierarchy only load data from off chip memory (HBM) when there is a cache miss. As HBM provides high bandwidth, we can prefetch some data that will be used in the near future to the cache to reduce memory stall. We can identify the memory load patterns using machine learning methods. After learning the pattern, we can use this pattern to guide data prefetch from the off-chip memory. This prefetch can significantly reduce memory stall time as the DRAM access time usually 100x times over the cache access.

- *Intelligent data management strategy on the GPU UVM.* Unified Virtual Memory (UVM) has drawn attentions from researchers as it greatly simplified the GPU programming efforts. It unifies the memory space of CPU

and GPU sides. The driver and the hardware will be responsible for the data migration. Current UVM management strategy is not efficient and results in significant performance degradation. Machine learning methods is an interesting topic that can be used to learn the data migration patterns between CPUs and GPUs. Once the pattern is identified, the machine learning agent can fetch the data from the CPU or evict data from the GPU side in advance to overlap the data migration latency. This research topic can be applied upon the Chapter 5 to further improve UVM performance.

Bibliography

[1] Fermi white paper.

[2] Radeons next-generation vega architecture, 2017.

[3] Cuda toolkit, 2020.

[4] 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.

[5] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 607–618, 2015.

[6] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[7] 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. ACM, 2017.

[8] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ACM SIGPLAN Notices*, volume 53, pages 503–518. ACM, 2018.

[9] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287. IEEE, 2017.

[10] Ali Bakhoda, George L. Yuan, Wilson WL Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, 2009.

[11] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[12] Thomas W Barr, Alan L Cox, and Scott Rixner. Spectlb: a mechanism for speculative address translation. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 307–318. ACM, 2011.

[13] Bradford M Beckmann and Anthony Gutierrez. The amd gem5 apu simulator: Modeling heterogeneous systems in gem5. *MICRO Tutorial*, 2015.

[14] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[15] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 383–394, 2013.

[16] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level tlbs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 62–63. IEEE, 2011.

[17] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40. IEEE, 2009.

[18] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. *ACM Sigplan Notices*, 45(3):359–370, 2010.

[19] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *IISWC*, 2012.

[20] Kevin K. Chang and et al. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *SIGMETRICS*, 2016.

[21] Amit S Chavan, Kartik R Nayak, Keval D Vora, Manish D Purohit, and Pramila M Chawan. A comparison of page replacement algorithms. *International Journal of Engineering and Technology*, 3(2):171, 2011.

[22] 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 *IISWC*, 2009.

[23] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IISWC*, 2010.

[24] Steven Chien, Ivy Peng, and Stefano Markidis. Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019.

[25] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[26] Intel Core. i7 processor series datasheet, 2010.

[27] Hongwen Dai, Christos Kartsaklis, Chao Li, Tomislav Janjusic, and Huiyang Zhou. Racb: Resource aware cache bypass on gpus. In *SBAC-PADW*, 2014.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[29] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel computer organization and design*. cambridge university press, 2012.

[30] KI Farkas and NP Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Intl. Symp. on Computer Architecture (ISCA)*, Apr. 1994.

[31] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[32] Debashis Ganguly, Z Zhang, J Yang, and Rami Melhem. Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In *Proc. of the Int. Conf. on Parallel and Distributed Processing (IPDPS)*.

[33] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2019.

[34] Víctor García-Flores, Eduard Ayguade, and Antonio J Peña. Efficient data sharing on heterogeneous systems. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 121–130. IEEE, 2017.

[35] Xun Gong, Rafael Ubal, and David Kaeli. Multi2sim kepler: A detailed architectural gpu simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–278. IEEE, 2017.

[36] 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.

[37] Cary Gray and David Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, volume 23. ACM, 1989.

[38] Khronos Group. *OpenCL*.

[39] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

[40] Minwoo Gu, Younghun Park, Youngjae Kim, and Sungyong Park. Low-overhead dynamic sharing of graphics memory space in gpu virtualization environments. *Cluster Computing*, pages 1–12, 2019.

[41] Yongbin Gu and Lizhong Chen. Cart: Cache access reordering tree for efficient cache and memory accesses in gpus. In *Intl. Conf. on Computer Design (ICCD)*, Oct 2018.

[42] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *ISPASS*, 2012.

[43] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun. 2009.

[44] Magnus Jahre and Lasse Natvig. Performance effects of a cache miss handling architecture in a multi-core processor. 2007.

[45] Magnus Jahre and Lasse Natvig. A high performance adaptive miss handling architecture for chip multiprocessors. In *Transactions on High-Performance Embedded Architectures and Compilers IV*. 2011.

[46] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. *CoRR*, abs/1901.05758, 2019.

[47] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA)*, 2014.

[48] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. An energy-efficient and scalable edram-based register file architecture for gpgpu. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun. 2013.

[49] Onur Kayıran, Adwait Jog, and Mahmut Taylan Kandemirand Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *PACT*, 2013.

[50] Jens Kehne, Jonathan Metter, and Frank Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 65–77, 2015.

[51] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: an extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[52] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. Efficient utilization of gpgpu cache hierarchy. In *Workshop on General Purpose Processing using GPUS (GPGPU)*, Feb. 2015.

[53] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.

[54] Youngsok Kim, Jaewon Lee, Jae-Eon Jo, and Jangwoo Kim. Gpudmm: A high-performance and memory-oblivious gpu architecture using dynamic memory management. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[55] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Intl. Symp. on Computer Architecture (ISCA)*, May. 1981.

[56] Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *High Performance Computer Architecture (HPCA)*, 2012.

[57] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun. 2015.

[58] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: enabling energy optimizations in gpgpus. In *ISCA*, 2013.

[59] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic gpu cache bypassing. In *International Conference on Supercomputing(ICS)*, 2015.

[60] 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. ACM, 2019.

[61] Lingda Li, Ari B. Hayes, Shuaiwen Leon Song, and Eddy Z. Zhang. Tag-split cache for efficient gpgpu cache utilization. In *International Conference on Supercomputing(ICS)*, 2016.

[62] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating opt with statistical clairvoyance and variable size caching. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256. ACM, 2019.

[63] Gabriel H Loh. 3d-stacked memory architectures for multi-core processors. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun. 2008.

[64] Qiumin Lu, Jianguo Yao, Haibing Guan, and Ping Gao. gqos: A qos-oriented gpu virtualization with adaptive capacity sharing. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):843–855, 2019.

[65] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[66] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426. IEEE, 2018.

[67] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[68] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 2009.

[69] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture(MICRO)*, 2007.

[70] Inc NanGate. Nangate freepd45 open cell library. *Avalible at: http://http://www.nangate.com/*, 2017.

[71] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *MICRO*, 2011.

[72] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *International Symposium on Microarchitecture(MICRO)*, 2006.

[73] NVIDIA Nsight and Visual Studio Edition. 3.0 user guide. *NVIDIA Corporation*, 2013.

[74] Nvidia. Nvidia geforce gtx 980 white paper, 2014.

[75] Nvidia. CUDA Runtime API – Memory Management, Aug 2019.

[76] GPU Nvidia. Computing sdk. *Gpu computing sdk, Avalible at: https://developer. nvidia. com/gpu-computing-sdk*, 2013.

[77] 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. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, MAR. 2014.

[78] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[79] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2013.

[80] Jason Power, Mark D Hill, and David A Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578. IEEE, 2014.

[81] Yu Qin. a stochastic decomposition implementation of support-vector machine training. `https://github.com/qin-yu/julia-svm-gpu-cuda`, 2019.

[82] Moinuddin K Qureshi, David Thompson, and Yale N Patt. The v-way cache: demand-based associativity via global replacement. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun. 2005.

[83] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 18. IEEE Press, 2016.

[84] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[85] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA*, 2000.

[86] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *International Symposium on Microarchitecture(MICRO)*, 2012.

[87] Nikolay Sakharnykh. Unified memory on pascal and volta, May 2017.

[88] Nikolay Sakharnykh. Beyond gpu memory limits with unified memory on pascal, Dec 2018.

[89] Ankit Sethia, D. Anoushe Jamshidi, and Scott Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *High Performance Computer Architecture (HPCA)*, 2015.

[90] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. Scheduling page table walks

for irregular gpu applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 180–192. IEEE, 2018.

[91] 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*, 2012.

[92] 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.

[93] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. Adaptive gpu cache bypassing. In *Workshop on General Purpose Processing using GPUS (GPGPU)*, Feb. 2015.

[94] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *MICRO*, 2006.

[95] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171. IEEE, 2016.

[96] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[97] Bin Wang, Yue Zhu, and Weikuan Yu. Oaws: Memory occlusion aware warp scheduling. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Sep. 2016.

[98] Yu Wang, Weikang Qian, Shuchang Zhang, Xiaoyao Liang, and Bo Yuan. A learning algorithm for bayesian networks and its efficient implementation on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):17–30, 2015.

[99] Steven JE Wilton and Norman P Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits (JSSC)*.

[100] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *PPoPP*, 2013.

[101] Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9621–9630, 2019.

[102] Intel Xeon. E5 processor series datasheet, 2012.

[103] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on gpus. In *ICCAD*, 2013.

[104] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for gpus. In *High Performance Computer Architecture (HPCA)*, 2015.

[105] 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.

[106] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, 2011.

[107] Wei Zhang, Mingfa Zhu, Tao Gong, Limi Xiao, Li Ruan, Yiduo Mei, Yuzhong Sun, and Xu Ji. Performance degradation-aware virtual machine live migration in virtualized servers. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 429–435. IEEE, 2012.

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