Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription

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

Ziyu Zhang
Department of Computer Science
University of Pittsburgh
ziz41@pitt.edu

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

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

***Abstract*—Unified Memory in heterogeneous systems serves a wide range of applications. However, limited capacity of the device memory becomes a first order performance bottleneck for data-intensive general-purpose applications with increasing working sets. The performance overhead under memory oversubscription depends on the memory access pattern of the corresponding workload. While a regular application with sequential, dense memory access suffers from long latency write-backs, performance of a irregular application with sparse, seldom access to large data-sets degrades due to page thrashing. Although smart spatio-temporal prefetching and large page eviction yield good performance in general, remote zero-copy access to host-pinned memory proves to be beneficial for irregular, data-intensive applications. Further, new generation GPUs introduced hardware access counters to delay page migration and reduce memory thrashing. However, the responsibility of deciding what strategy is the best fit for a given application relies heavily on the programmer based on thorough understanding of the memory access pattern through intrusive profiling. In this work, we propose a programmer-agnostic runtime that leverages the hardware access counters to automatically categorize memory allocations based on the access pattern and frequency. The proposed heuristic adaptively navigates between remote zero-copy access to host-pinned memory and first-touch page migration based on the trade-off between low latency remote access and high-bandwidth local access. We show that although designed to address memory oversubscription, our scheme has no impact on performance when working sets fit in the device-local memory. Experimental results show that our scheme provides performance improvement of 22% to 78% for irregular applications under 125% memory oversubscription compared to the state of the art. At the same time, regular applications are not impacted by the framework.**

***Keywords*-page migration, pinning, memory management, CPU-GPU, Unified Memory**

I. Introduction

Energy efficiency and massive data parallel SIMD nature of GPU architecture have led to wider adoption of GPUs by general purpose applications [10], [24]. Traditionally, these regular applications operate on highly-structured large vectors in a streaming fashion. However, in recent years, there has been an increasing trend to use GPUs for applications with irregular memory access patterns, such as data mining, social network analysis and bioinformatics. These algorithms operate on large, irregular data structures like trees, and graphs and are highly input dependent. They exhibit statically unpredictable, memory access irregularity, and consequently low spatial locality.

Because of their dense, sequential memory access, *regular* data-parallel applications benefit from prefetchers [13], [25], [29]. A prefetcher prefetches data in advance based on spatio-temporal locality of access. In the process, it reduces the number of faults and further improve PCI-e bandwidth. However, for *irregular* applications, aggressive prefetching can be counter-productive under memory oversubscription. The situation is aggravated further as heavily referenced pages are replaced using LRU without differentiating between cold and hot data structures.

Usage of host-pinned "Zero-copy" memory buffers is suggested in both CUDA [19] and OpenCL [3], [5] for irregular applications with sparse, rare access to large data. Using remote zero-copy has two advantages: (i) as no data is copied to the device memory, it prevents memory oversubscription, and (ii) sparse accesses benefit from low latency direct access. In newer generation GPUs [26], page-level access counters are used to delay migration of pages from preferred location to the local memory. Avoiding first touch migration helps reduce page thrashing for irregular applications. However, there is a drawback with remote "Zero-copy" access and delayed migration. If GPU reads or writes to host pinned memory directly for multiple times, then kernel (GPU code) execution will slow down. Particularly, *regular* applications with dense, sequential memory access will no longer benefit from bandwidth-optimized local access that would result after migrating the data from host memory. Consequently, developers resort to extensive

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00054

memory profiling before specifying any hint to the memory subsystem or deciding on a particular allocation type.

Researchers [1], [2] have studied the trade-offs between lower latency direct memory access to host pinned memory and bandwidth optimized local memory access in the context of heterogeneous systems. These works try to split pages between host and device memory based on the ratio of host-to-device interconnect bandwidth and local memory bandwidth to fully utilize the overall system-wide memory bandwidth. Further, prior work [2] proposed static page placement strategies, which require compiler support for profiling and programmer intervention to annotate data structures. NVIDIA CUDA runtime also supports placement of memory pages to either host or device memory based on user-provided hints. If no API annotation is provided, then CUDA runtime maintains lists of pages thrashed and pinned indexed by the host or GPU identifier and throttles page migration and prefetch decision for these pages. Firstly, maintaining list of pages grouped in blocks of virtual pages has a considerable implementation and space overhead. Secondly, page-wise throttling decision eliminates the benefits of prefetching and eviction in bulk [13].

We provide extensive analysis of memory access pattern of popular GPU workloads to classify them under *regular* and *irregular* categories. Further analysis reveals that working sets of *irregular* applications can often be divided into *cold* and *hot* data structures. While *cold* allocations are accessed seldom and sparsely, *hot* data structures have dense sequential access.

To this end, we propose a dynamic page placement strategy for *irregular* general purpose applications. Our proposed framework leverages hardware-based access counters [25] to identify sparse and dense memory access and differentiate between *hot* and *cold* allocations. Over the course of execution, the framework achieves a balance between low latency remote access to host-pinned *cold* allocations and bandwidth-optimized local access to *hot* allocations in *irregular* applications with oversubscribed working sets. However, our framework does not affect *regular* applications and applications with working sets smaller than the device memory capacity. Through experimentation with a set of *regular* and *irregular* GPGPU applications, we demonstrate that the proposed heuristic adaptively navigates the spectrum between zero-copy remote access and first-touch migration. The key contribution of this work is leveraging existing system support to design such a developer-agnostic framework. Specifically, the proposed framework is not built on any new hardware modification and does not require any explicit user hints that are based on intrusive profiling of workloads to provide performance improvement for *irregular* applications. Moreover, the proposed framework builds on generic concepts like zero-copy memory and delayed migration and thus can be adopted by any GPU irrespective of the vendor-specific architecture and runtime.

II. Background

This section discusses on-demand paging, the tree-based prefetcher and the page replacement policy key in realizing Unified Memory for discrete CPU-GPU heterogeneous systems. Though the description closely follows NVIDIA/CUDA terminology, the high-level concepts are generalizable and vendor-agnostic.

A. On-demand Page Migration and Unified Memory

In the classic "copy then execute" model, data shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the host program before and after the kernel launch. NVIDIA Pascal GPUs [20] have introduced hardware page faulting and Page Migration Engine to support Unified Memory for discrete CPU-GPU systems. In CUDA 8.0 [19], `cudaMallocManaged` allows programs to allocate data that can be accessed by both host code and kernel using a single shared pointer. The illusion of Unified Memory is realized by on-demand allocation and fault-driven data transfer. This improves programmablity.

In the "copy then execute" model, the host program ensures that data is physically available in the device memory before kernel starts executing. Warps are stalled on *near-faults* which occurs only upon L2 cache misses. The massive thread-level parallelism (TLP) hides the local memory access latency and guarantees high throughput. However, in Unified Memory, a new type of faults, which we will refer to as *far-faults* [29], can occur when data is not physically present in the device local memory. On-demand allocation and page migration is triggered by these far-faults. As a result, a far-fault is much costlier than a near-fault. The overhead of a far-fault consists of two major components: a far-fault handling latency (typically $45\mu s$ in Pascal GPUs) to walk and manage page table and the data migration latency over PCI-e interconnect. The data migration and kernel execution is serialized.

B. Tree-based Prefetcher

In Unified Memory, massive TLP is not sufficient to mask memory access latency as the offending warps stall for the costlier far-faults. The total kernel execution time increases dramatically and closely resembles the serialized data migration and kernel execution time of the "copy then execute" model. To ameliorate this situation, CUDA 8.0 introduced `cudaMemPrefetchAsync` which allows programmers to overlap the kernel execution with asynchronous parallel data migration. However, the onus of deciding what and when to prefetch still lies on the programmers. To address this challenge, prefetchers have been proposed [29] and shown to provide dramatic performance improvement over replayable far-fault based page migration. Agarwal et al [1] proposed prefetching neighbors of touched pages to reduce the overhead of shared TLB shootdown.

Ganguly et al [13] uncovered the semantics of a "tree-based neighborhood prefetcher" implemented by CUDA runtime. They show that this prefetcher provides the best performance compared to other prefetchers [1], [29] proposed earlier and is key to the success of Unified Memory. We verified the mechanism of this prefetcher by running a set of published micro-benchmarks [11] on `GeForceGTX 1080 ti` [23]. We also confirmed the understandings by studying the MIT-licensed opensource `nvidia-uvm` submodule. The semantics of this tree-based prefetcher is described below.

Upon allocating data with `cudaMallocManaged`, the user-specified size is first rounded up to the next $2^i * 64KB$. Then, the allocated size is logically divided into $2MB$ large pages plus a fraction of $2MB$. For example, from a user specified size of $4MB + 168KB$, three logical chunks are created- two chunks each of $2MB$ and one of $256KB$. Then each of these chunks are further divided into $64KB$ basic blocks, which is the unit of prefetching, to create three full-binary trees where leaf-levels hold $64KB$ basic blocks.

On every first-touch to a page, a $64KB$ basic block at leaf level is identified for migration. As the leaf-levels are populated by on-demand allocation and data migration, the occupancy levels in the tree is updated starting from the leaf to the root level. At any point, if runtime identifies that occupancy of any non-leaf level node is strictly more than 50%, a prefetch decision is made and pushed to the leaf levels to balance the occupancy between two children of the node. In this process, one or many empty leaf node(s) of $64KB$ basic block are identified for prefetching. We refer the readers to the work [13] for detailed heuristic of this tree-based prefetcher.

Typical GPGPU workloads are massively parallel and show spatio-temporal locality. Thus, the tree-based prefetcher, upon limiting its prefetch decision within $2MB$, provides spatio-temporal locality within large pages. Moreover, it results in allocation of contiguous physical memory and thus helps reduce bypassing nested page table walk. Tree-based prefetcher trades in the spectrum of two extremities: $4KB$ small page and $2MB$ large page. It adapts to the current state of the tree and opportunistically decides on the prefetch size ranging from $64KB$ to $1MB$ instead of a fixed granularity.

C. Page Replacement Policy

One of the major benefits of Unified Memory is that programmers do not need to worry about the size of working set and the available device memory space. When the working set of the GPU applications does not fit in the device memory, older pages are automatically evicted to make room for newer page migration. NVIDIA GPUs implement Least Recently Used (LRU) based page replacement policy. As the pages are migrated in, they are placed in a queue based on the migration timestamp. After migration if a page is accessed, then its position is updated based on the current access timestamp. Newly accessed pages are moved to the end of the queue and thus the oldest accessed (/migrated) page will be evicted upon oversubscription. This is how LRU page replacement policy is realized in NVIDIA GPUs. The page replacement works at the strict granularity of $2MB$ large page. A $2MB$ large page is selected for eviction only when it is fully populated and not currently addressed by scheduled warps. Evicting $2MB$ ensures that the semantics of the tree-based prefetcher is not violated. Hence, the prefetcher remains in action even after device memory oversubscription [13].

III. Challenges with Irregular Applications under Oversubscription

In this section, we demonstrate the challenges with memory oversubscription and how a prefetcher exacerbates the problem. Further, we characterize GPGPU workloads analyzing their memory access pattern to motivate our work. Lastly, we detail the state of the art GPU programming practices like zero-copy memory access and access counter based delayed migration and highlight limitations with these approaches.

A. Oversubscription Overhead

While the tree-based prefetcher improves performance by reducing the number of far-faults and provides higher programmability, aggressive prefetching under memory oversubscription proves to be counter productive. A prefetcher prefetches pages with spatio-temporal locality with the anticipation that these pages will be consumed by the threads in the immediate future. However, aggressive prefetching under memory constraint can cause displacement of heavily referenced pages. As a result, GPGPU workloads suffer from dramatic performance degradation.

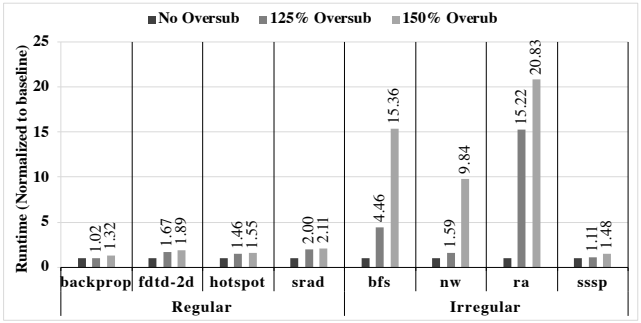


Figure 1: Sensitivity of workloads to the percentage of memory oversubscription (performed on real hardware).

Figure 1 shows the performance degradation of GPGPU workloads with varied percentage of memory oversubscription [1]. The results are obtained by running the workloads

[1] these workloads are described in Section V

on GeForceGTX 1080 ti [23] *(not on simulated environment)*. To emulate memory oversubscription, working sets of the workloads are not scaled, rather the total free space is controlled by allocating dummy cudaMalloced variables because cudaMalloced allocations are pinned and not selected for eviction.

Further analysis by runtime profiling shows that fault-based migration under oversubscription waits for long latency write backs in case of *regular* applications. On the other hand, the oversubscription overhead in *irregular* applications is due to excessive page thrashing which is further exacerbated by the prefetcher. *Irregular* applications show order of magnitude performance degradation.

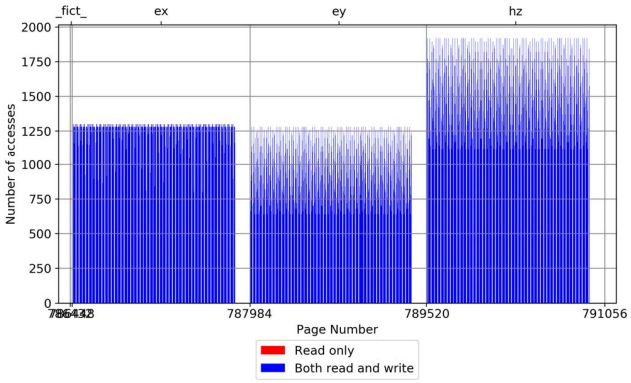
B. Workload Characterization

An effective memory management strategy to deal with device memory oversubscription requires thorough understanding of the memory access pattern of the workloads. To this end, we analyze the memory access pattern of various GPGPU workloads to characterize their respective behaviour. We find that workloads can be broadly categorized into: 1) *regular* with dense, sequential, repetitive memory access and 2) *irregular* with sparse, seldom access.

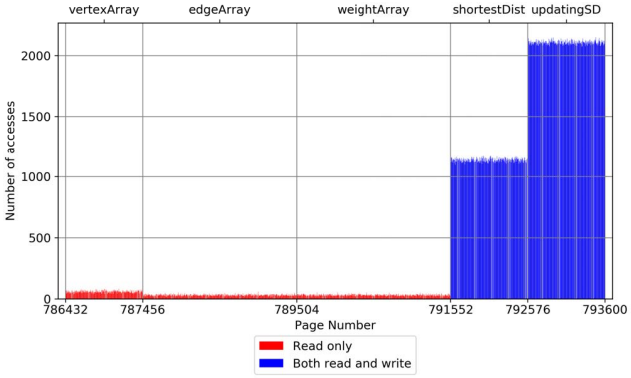
Firstly, we visualize the distribution of page access frequency of different data structures over the entire execution period of two benchmarks, fdtd and sssp, in Figure 2. We characterize memory pages based on the type of access - *read only*, and *both read and in-place write*.

Figure 2a shows that in fdtd most of the pages in the allocations are accessed at the same frequency over the entire execution time. A very few pages equally spaced over the allocation boundary are accessed a lot more than the rest of the pages. On the other hand, Figure 2b shows an entirely different characteristics for sssp. We can see that few data-structures are more heavily accessed than the others leading to a cluster of *hot* and *cold* pages over the entire memory set. Moreover, the read-only data-structures are *cold* and the pages in *hot* data-structures are both read from and written to. This shows that for *irregular* applications a small fraction of memory footprint corresponds to the higher share of bandwidth.

Figure 3a, and 3b show the memory access pattern of fdtd in iterations 2, and 4 respectively. Horizontal axis represents time in cycles and primary vertical axis shows the page numbers accessed at a given cycle. We can see that the memory access pattern is fairly constant over two different iterations. Moreover, in every iteration, each allocated data structure is accessed linearly. This explains the access frequency distribution of fdtd in Figure 2a. We characterize fdtd as a *regular* application. Regular applications typically show dense, sequential access repeated over multiple iterations. backprop, hotspot, and srad are other examples of GPU benchmarks that can also be



(a) fdtd



(b) sssp

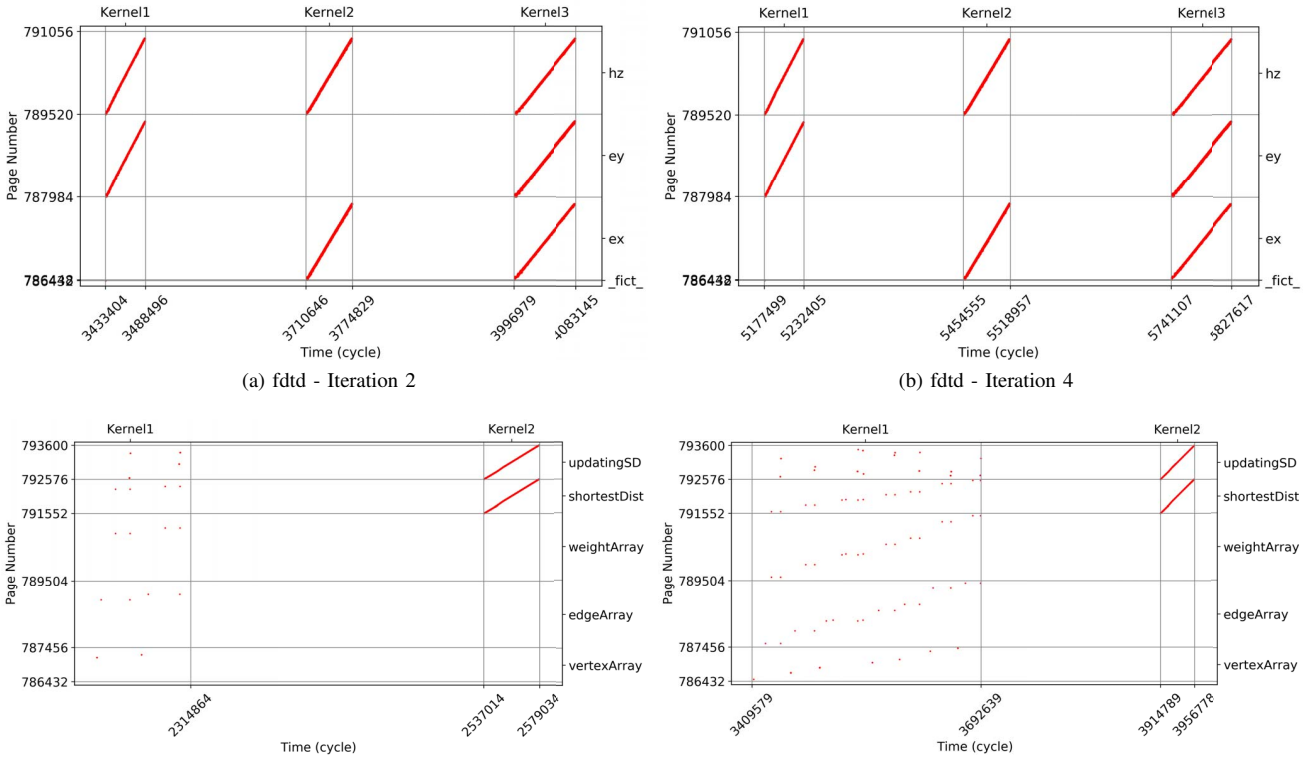
Figure 2: Visualizing page access distribution detailing type of access and total number of accesses per page per managed allocation for fdtd and sssp.

categorized as *regular* applications as they exhibit similar memory access pattern [9].

On the other hand, Figure 3c, and 3d show the memory access pattern of sssp in iterations 3, and 5 respectively. We can see that kernel1 exhibits sparse memory access over different data structures and the memory pages accessed over different iterations varies drastically in virtual address space. However, kernel2 shows sequential and dense access over two data structures in every iteration. This justifies the cluster of *hot* and *cold* data structures in sssp as shown in Figure 2b. Hence, we characterize sssp as an irregular applications. In general, *irregular* applications exhibit dense sequential access on *hot* data structures and sparse, random access on *cold* data structures. bfs, nw [9], ra [27] are other benchmarks that fall under the same category.

C. Remote Zero-copy Access and Delayed Migration

Unified Memory offers a "single-pointer-to-data" model. Both host and device see a unified view of virtual address space. At any given time, only one physical copy of the data is maintained either on the host or the device memory.



(a) fdtd - Iteration 2

(b) fdtd - Iteration 4

(c) sssp - Iteration 3

(d) sssp - Iteration 5

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

Typically, the data is initialized in the host memory. On every first access to a page by the device, the corresponding page table entry in the host is invalidated and data is migrated to the device memory and a new entry is created in the device page table. On the contrary, with zero-copy allocations the physical allocation is hard-pinned to the host memory. This means pages are never copied from host to device memory. Rather the device accesses data remotely over cache-coherent interconnect. `cudaHostRegister` API allows `malloc`ed allocation to be pinned to the host memory and the kernels are launched with device pointer derived using `cudaHostGetDevicePointer` API. Remote zero-copy access has lower latency than the classic Direct Memory Access (DMA), but also suffers from lower bandwidth of PCIe interconnect. This is why zero-copy access is introduced for applications with seldom and sparse access to very large data sets. OpenCL [3], [5] also provides support for allocating host pinned memory using `CL_MEM_ALLOC_HOST_PTR`.

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

NVIDIA Volta GPUs [26] and IBM Power9 [15] introduced a new hardware based page-level access counter. If an allocation is advised to be soft-pinned to the host memory, then the memory is not copied directly at the first-touch by the device. Rather, the migration from the preferred location of host memory to the device memory is delayed based on a static access counter threshold, t_s. If the page is accessed to read data for a certain number of times crossing the value of t_s configured in the driver, the data is copied to the device memory. On the other hand, on write access, the page is invalidated in the host page table and exclusively copied to the device memory irrespective of the access frequency [25].

Irregular applications with sparse memory access can highly benefit from both remote zero-copy access and access counter based delayed migration. As in Unified Mem-

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

IV. Dynamic Access-counter Threshold Based Delayed Data Migration

We motivate our work based on the following observations: ❶ a higher percentage of PCIe bandwidth is consumed by a small percentage of the total memory pages, ❷ migrating pages of *cold* data structures causes eviction of pages of *hot* data structures for *irregular* applications, ❸ data migrations due to page thrashing over low bandwidth PCIe contribute to memory oversubscription overhead, ❹ current state-of-the-art solutions are not satisfactory to all workloads as zero-copy access and delayed migration can hurt performance of regular applications although proven to be useful for irregular workloads, and thus ❺ an effective solution to address device memory oversubscription must rely on user-hints based on extensive recognition of memory usage and access pattern.

In this section, we propose an adaptive runtime heuristic that is programmer-agnostic as it requires no advise to the memory subsystem from the application developer. Further, we leverage the new hardware features of page-level access counters to build our solution. Thus, it demands no hardware modification and is solely based on pragmatic modification to GPU driver.

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

So, an effective solution should be able to decide how to eliminate the overhead of remote access for no memory oversubscription and regular applications in general. We propose a dynamic threshold for delayed migration. The heuristic of our proposed solution is driven by the following equation:

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

where $t_s =$ Static access counter threshold,

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

$p =$ Multiplicative Migration Penalty

The proposed dynamic threshold, t_d, grows adaptively in response to the size of free space in the device memory starting from 1 to the driver configured static threshold. Let us consider the static threshold, t_s, configured in the driver as 8. If currently less than 12.5% of device memory is allocated, then the dynamic threshold is derived as 1 from Equation 1. This means every first touch will cause page migration. Similarly, the dynamic access counter threshold will be same as the static threshold of 8 just before reaching the full capacity of device memory and 9 upon oversubscription. The goal of the framework, here, is to tame down the aggression of the prefetcher by delaying the page migration as the memory starts filling up to its maximum capacity. Use cases involving no memory oversubscription and regular applications benefit from this mechanism compared to delayed migration based on static threshold.

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

Access Counter Based Page Replacement. The framework also extends page replacement strategy leveraging the same access counters. As detailed in Section II-C, a naïve LRU page replacement cannot differentiate a set of *cold* pages from a set of *hot* pages. As a result, it may end up evicting highly referenced *hot* pages in the process of migrating a *cold* page and thus defeats the objective of hard-pinning hot pages to the device memory and cold pages to the host memory. We use the access counters to sort the list of $2MB$ large pages in LRU list such that cold pages are prioritized over hot pages for eviction in irregular applications. Thus, we incorporate a simplified Least Frequently Used (LFU) scheme in the framework. However, with linear sequential access in regular applications, where pages are accessed with almost the same frequency, our framework automatically falls back to the LRU policy. We also prioritize read-only pages as eviction candidates. This is because on write access hot pages are migrated exclusively to the device memory irrespective of their access counter. So, we would prefer to keep the write pages in local memory as much as possible.

Access Counter Granularity. Access counters are maintained at the page granularity for Volta GPUs [26]. However, as explained in Section II-B, the tree-based prefetcher in `nvidia-uvm` module migrates data in multiple of $64KB$ basic blocks based on the page faults relayed from GMMU. This leads us to the optimization of maintaining access counters at $64KB$ basic block level instead of $4KB$ page granularity. This not only reduces the memory overhead of maintaining access counters, it is also functionally more meaningful as the prefetch granularity is $64KB$.

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

V. Experimental Methodology

Simulation Framework. Ganguly et al [13] extended GPGPU-Sim 3.x [6] to provide functional and timing simulation support for Unified Memory such that benchmarks written using `cudaMallocManaged`, and `cudaDeviceSynchronize` APIs can be simulated. They incorporated the control flow to resolve far-faults as described by Zheng et al [29]. This enables modelling on-demand memory allocation and fault-driven data migration. They also incorporated several prefetch mechanisms [29] including NVIDIA driver's tree-based prefetcher and different page replacement algorithms including default $2MB$ LRU scheme. The timing simulation of the setup is validated against a hardware platform with `GeForceGTX 1080 Ti` and `PCI-e 3.0 16x` interconnect.

We extended GPGPU-Sim UVM Smart to add support for direct access to host-pinned memory over PCIe interconnect and the LFU eviction policy. We also incorporated the access counter based automatic categorization of data structures as *cold* and *hot*, and adaptive dynamic threshold based page migration scheme as described in Section IV. Table I shows the primary configuration parameters of the extended simulator. The items in bold face shows the default value of the configurations items. The extended simulator is publicly available [11] for verification and further research collaboration.

Simulator	GPGPU-Sim UVM Smart
GPU Architecture	NVIDIA GeForceGTX 1080Ti Pascal-like
GPU Cores	28 SMs, 128 cores each @ 1481 MHz
Shader Core Config	Max. 32 CTA and 64 warps per SM, 32 threads per warp, GTO scheduler
Memory System	
Page Size	4KB
Page Table Walk Latency	100 core cycle
CPU-GPU Interconnect	PCI-e 3.0 16x, 8 GTPS per channel per direction, 100 GPU core cycles latency
DRAM Latency	100 GPU core cycles [2]
Remote Zero-copy Access Latency	200 GPU core cycles
Eviction Granularity	2 MB, 64KB
Page Replacement Policy	LRU, **LFU**
Far-fault Handling Latency	$45\mu s$
Hardware Prefetcher	Tree-based
Static Access Counter Threshold	**8**, 16, 32
Multiplicative Migration Penalty	2, 4, **8**, 1048576

Table I: Configuration parameters of the simulated system.

Application Suite. Along with the simulation framework, GPGPU-Sim UVM Smart [11] also provides a set of benchmarks from Rodinia [9], Lonestar [7], and Poly-Bench [14] benchmark suites. These are the only publicly available benchmarks that are implemented using CUDA Unified Memory APIs. In these benchmarks data structures are allocated using `cudaMallocManaged` instead of `cudaMalloc` followed by `cudaMemcpy` API calls.

As discussed in Section III-B, we divide these benchmarks into two categories: regular (`backprop`, `fdtd`, `hotspot`, `srad`), and irregular (`bfs`, `nw`, `ra`, `sssp`) access pattern.

To simulate oversubscription, working sets of the benchmarks are not scaled, rather is controlled by a configuration parameter that specifies the available free space in the device memory in the simulation setup.

VI. Evaluation

In this section, we evaluate the effectiveness of the framework proposed in Section IV. Henceforth, we will alternately refer to the dynamic access-counter threshold based delayed migration scheme as *Adaptive*. We compare our scheme with ❶ the state of the art baseline where remote access is not enabled and data is migrated at first touch, alternately referred it as *Baseline* or *Disabled*, ❷ the static access counter based threshold proposed in Volta GPUs termed as *Always*, and ❸ a static access counter based delayed migration scheme enabled only after oversubscription referred as *Oversub*. Difference between *Always* and *Oversub* is that *Always* delays migration from the start irrespective of memory oversubscription.

For *Baseline*, LRU page replacement is active whereas for the other three scheme, our framework uses the proposed access counter based simplified LFU policy.

Note that the following experiments only deal with 125% of device memory oversubscription. Unlike, CPU virtual memory, current GPUs are not capable of handling a higher percentage of memory oversubscriptions. NVIDIA recommends to use more than one GPU to distribute workload if the GPU memory oversubscription is more than 125%.

A. Sensitivity to Static Migration Threshold

The success of access counter based delayed migration relies on finding a suitable value for the static access counter threshold, t_s. The objective of the framework is not to hurt *regular* applications in general and all applications under no oversubscription. So, we run experiments to find out the sensitivity of t_s on the kernel execution time. Figure 4 shows the result. In this experiment, we consider the *Always* scheme as it is the state of the art for delayed migration.

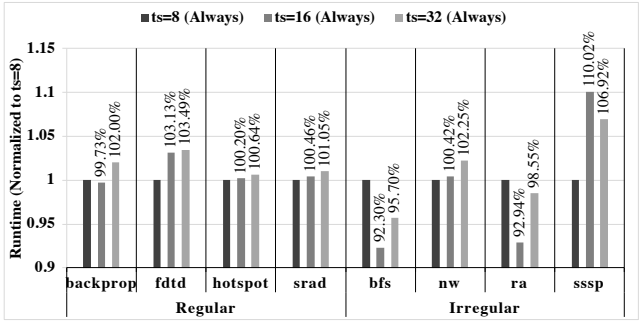


Figure 4: Sensitivity of workloads to the static access counter threshold for delayed migration.

As we vary t_s from 8 to 32, we see that *regular* applications show almost no sensitivity to the static threshold. This is because for regular applications the number of per basic block accesses generated by load/store unit is quite high and they always exceeds the threshold. Thus, for *regular* application, no remote access is performed. However, *irregular* applications shows sensitivity to t_s. While performance of `nw` and `sssp` degrades with higher value of t_s, `bfs` and `ra` show improvement for $t_s = 16$ compared to $t_s = 8$. This behaviour is not unpredictable and depends on the input of the workload and the sparsity of memory access. We recommend a justifiably small number for t_s such that it closely resemble to first-touch migration under no oversubscription. However, an extremely small value for t_s like 1 or 2 is also not recommended as it will have negative impact on performance for irregular applications under oversubscription. Experiments in the later subsections use $t_s = 8$. We will present experimental results showing the sensitivity p on performance in Section VI-D.

B. The Case of No Oversubscription

In this section, we compare the proposed *Adaptive* scheme against the *Baseline* and the *Always* scheme for delayed migration under no oversubscription. Figure 5 shows the normalized runtime of different workloads using the above three schemes. *Oversub* is not applicable for this experiment as it enables threshold based delayed migration only after oversubscription.

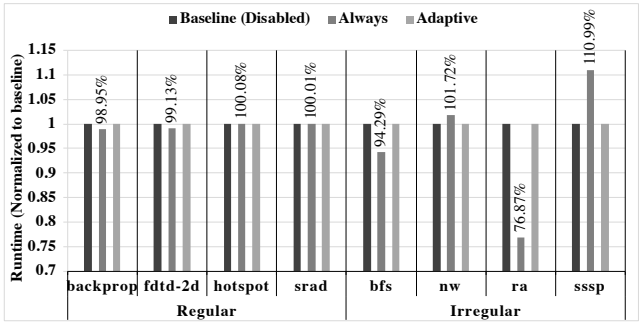


Figure 5: Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration scheme under no memory oversubscription.

Figure 5 shows for both *regular* and *irregular* applications, the *Adaptive* scheme produces results equivalent to the *Baseline* or *Disabled* scheme. This means that the dynamic threshold scheme falls back to first-touch migration based on the access frequency and memory availability. While, for *regular* applications, *Always* scheme shows no major performance difference, for *irregular* applications, it introduces unpredictability. `bfs` and `ra` benefit from *Always* scheme, whereas `nw` and `sssp` show performance degradation. This is because even for sparse memory access,

if there is no memory constraint, it is always better to copy the data to device memory using prefetcher and then benefit from bandwidth optimized local access. Note, that the objective of the framework under no oversubscription is not to outperform *Baseline* first touch-based migration, rather to show more consistent and predictbale behavior compared to *Always* scheme of static threshold-based delayed migration.

C. The Case of Oversubscription

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

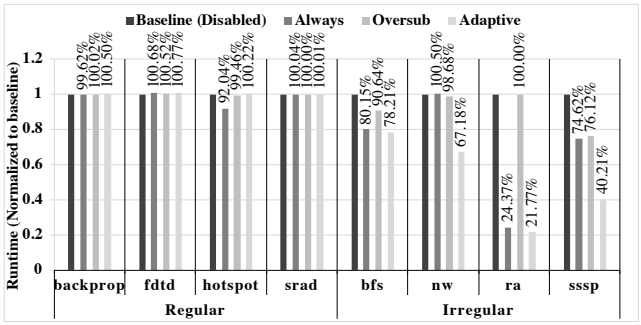


Figure 6: Comparing the impact of dynamic access counter based adaptive scheme on execution time against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes.

Figure 6 shows that *Adaptive* scheme does not impact performance of *regular* applications. On the other hand improves the performance of *irregular* applications by 22% to 78%. Moreover, it also yields better performance compared to static access counter threshold based schemes.

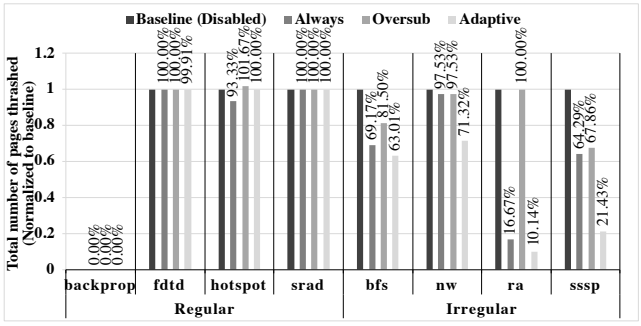


Figure 7: Comparing the impact of dynamic access counter based adaptive scheme on memory thrashing against the baseline case of first-touch migration and static access counter threshold based delayed migration schemes.

To reason about the performance improvement by the *Adaptive* scheme demonstrated in Figure 6, we plot the number of pages being thrashed for different schemes in Figure 7. We see that the improvement in kernel execution time is directly a factor of reduction in memory thrashing for *irregular* applications. For *regular* applications, the number of pages being thrashed using *Adaptive* scheme is same as *Baseline* or *Disabled*. Note that for `backprop` there is no thrashing at all. This is because it scans through the entire allocation sequentially without any data reuse over iterations. On the other hand, `ra` shows completely random access and no data reuse which makes it a perfect candidate for zero-copy host-pinned memory access.

D. Sensitivity to Multiplicative Penalty

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

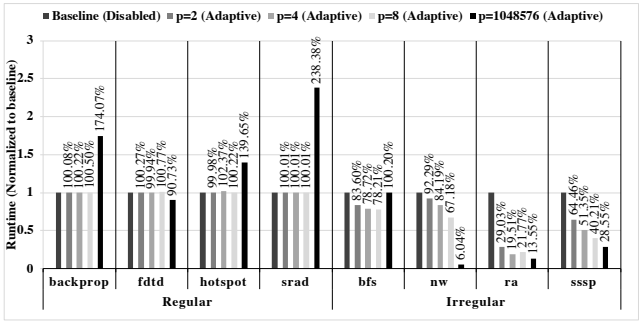


Figure 8: Sensitivity of workloads to the multiplicative migration penalty.

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

A question may arise as to why not having an unreasonably higher value of p. Clearly, the dynamic threshold, t_d is dictated by p. Hence, for relatively large p, the values of t_s and r would not have an appreciable effect on t_d. As a result, an unreasonably large p will blindly keep pages pinned to the host memory without caring for the access threshold or the number of round trips (evictions). We see that $t = 1048576$ indeed has a huge performance benefits on `nw`, `ra` and `sssp` by eliminating thrashing entirely. However, this behaviour is unpredictable and solely depends on what pages get pinned to the host memory. For example, `bfs` shows 2% performance degradation. Moreover, regular

applications suffer a great deal of performance loss for a large p. For example, kernel execution time for `srad` almost doubles up. This is because for dense, sequential access it is always better to migrate the memory to the device and access locally. This also proves that our framework is tunable to achieve remote zero-copy access by configuring p or multiplicative penalty. Further, our dynamic threshold based heuristic navigates between zero-copy remote access and first touch migration adaptively.

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

VII. Related Work

Unified Virtual Memory (UVM) support in modern discrete CPU-GPU systems [4], [20] has overcome many limitations present in the traditional "copy then execute" programming model [21], [22] by automating GPU memory management. Agarwal et al [1] proposed aggressive first-touch migration and prefetching neighboring pages. Zheng et al [29] are the first one to study different user-directed and user-agnostic prefetchers to overlap data migration and kernel execution to hide the overhead of handling far-faults. Ganguly et al [13] uncovered the mechanism of the tree-based prefetcher implemented in NVIDIA GPU dirver and demonstrated that compared to other prefetchers it provides the maximum performance improvement.

Burtscher et al [7] introduced a new benchmark suites called Lonestar consisting of a set of irregular applications. They performed a quantitative study to categorize these workloads based on their memory and control flow irregularity and input dependence. Pannotia [8] focused on evaluating irregular graph applications on AMD platform. Spatter [17] is a CUDA benchmark suite to characterize scatter, gather, and related sparse access patterns. Vesely et al [28] showed that divergent memory accesses in irregular application can cause order of magnitude slow down from address translation overhead alone. Leo [12] is a profiler driven optimization framework for irregular GPU applications.

Oversubscription overhead for data-intensive general purpose applications have become a first order performance bottleneck. Data-parallel workloads are partitioned in VAST [18] based on available GPU physical memory. In the contrary, GPUswap [16] transparently relocates data from the GPU to system RAM but make it accessible to the device under oversubscription. Agarwal et al [2] have proposed a compiler-based profiling mechanism to make programmers aware of the specific memory access pattern. Based on this knowledge, programmers annotate data structures to provide page placement hints to the the memory subsystem. All these techniques either are plagued by huge performance overhead or needs code modification. Ganguly et al [13] proposed a tree-based page replacement policy inspired by the tree-based prefetcher in CUDA driver to deal with memory oversubscription. However, they failed to address the specific case for irregular applications with sparse, seldom access to large oversubscribed datasets.

VIII. Conclusion

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

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-1725657. We thank the anonymous reviewers for their constructive feedback.

References

[1] 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.

[2] 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*, ASPLOS '15, pages 607–618, New York, NY, USA, 2015. ACM.

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

[4] AMD. Radeons Next-generation Vega Architecture. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf, 2017.

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

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

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

[8] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

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

[10] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and John Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference, number AIAA*, volume 4001, 2009.

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

[12] Naila Farooqui, Christopher J Rossbach, Yuan Yu, and Karsten Schwan. Leo: A profile-driven dynamic optimization framework for {GPU} applications. In *2014 Conference on Timely Results in Operating Systems ({TRIOS} 14)*, 2014.

[13] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 224–235, New York, NY, USA, 2019. ACM.

[14] 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.

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

[16] 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*, VEE '15, pages 65–77, New York, NY, USA, 2015. ACM.

[17] Patrick Lavin, Jason Riedy, Rich Vuduc, and Jeffrey Young. Spatter: A benchmark suite for evaluating sparse access patterns. *arXiv preprint arXiv:1811.03743*, 2018.

[18] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Vast: The illusion of a large memory space for gpus. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 443–454. IEEE, 2014.

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

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

[21] NVIDIA Corp. CUDA Toolkit 4.0. https://developer.nvidia.com/cuda-toolkit-40, 2011.

[22] NVIDIA Corp. NVIDIA GeForce GTX 750 Ti. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.

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

[24] Victor Podlozhnyuk. Black-scholes option pricing, 2007.

[25] Nikolay Sakharnykh. Everything you need to know about Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. Accessed Apr 04, 2019.

[26] Nikolay Sakharnykh. Unified memory on pascal and volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf. Accessed Apr 04, 2019.

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

[28] 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.

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