To Move or Not to Move? - Page Migration for Irregular Applications in Over-subscribed GPU Memory Systems with DynaMap

Chia-Hao Chang
Pennsylvania State University

Adithya Kumar
Pennsylvania State University

Anand Sivasubramaniam
Pennsylvania State University

Abstract

This paper focuses on the severe page thrashing problem that can arise when running large irregular memory access applications on limited GPU memory systems. Such memory over-subscription causes very poor performance in the currently on demand (eager) or page-group granularity access-counter based (lazy) page migration mechanisms found in NVIDIA's UVM drivers. Our detailed analysis of these executions reveals a very novel insight: rather than duplicate the responsibility of catering to both temporal and spatial locality in both GPU caches and its memory, it is better for the former to simply cater to the temporal aspect, and the latter to the spatial aspect, thereby saving precious memory system capacities. Based on this, we build an adaptive page migration scheme, called DynaMap, that (i) uses a compiler pass to instrument off-the-shelf CUDA UVM applications for spatial utilization tracking, (ii) dynamically sets a spatial utilization threshold to determine migration based on memory pressure and access characteristics, and (iii) enhances the current NVIDIA UVM driver to dynamically migrate the page (from the host memory to the GPU) based on the threshold. Using 7 irregular applications from public benchmark suites, we implement DynaMap on a real system with different over-subscription ratios to show speedups as much as 2.5× (34% on the average) over state-of-the-art UVM implementations.

CCS Concepts

• **Software and its engineering** → **Memory management**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Keywords

GPGPU; UVM; Memory Oversubscription

ACM Reference Format:

Chia-Hao Chang, Adithya Kumar, and Anand Sivasubramaniam. 2021. To Move or Not to Move? - Page Migration for Irregular Applications in Over-subscribed GPU Memory Systems with DynaMap. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463766

1 Introduction

The high throughput computational capabilities of GPUs have brought them into the mainstream to tackle the demands of applications from diverse domains including entertainment, bio-informatics, scientific computing, graph and other big data analytics. In order to sustain the data needs of numerous cores, GPU memory systems are highly tuned for high throughput, with specialized High Bandwidth Memories (HBM) used on these GPU cards. Such memories are designed and fabricated differently, to specific market segments, and are hence much more expensive per GB (2.5-3× more expensive [1]) compared to their traditional DDR counterparts, which are more commoditized today. Consequently, the memory capacities of the GPU cards are not quite sufficient to completely hold the entire datasets of many current and future big-data applications. Until recently, many of these applications, including the popular deep learning python libraries, have either (i) offloaded kernels to these GPUs that fit completely, and/or (ii) explicitly moved data back-and-forth between the GPU and CPU memories similar to how programs were written before the advent of virtual memory. To address these capacity limitations and programming interventions, Unified Virtual Memory (UVM) has been proposed to treat the union of GPU and host CPU memories as a single unit, with demand paging like mechanisms used to dynamically move pages back-and-forth. In this paper, we point out that page migration techniques in UVM implementations of current systems (like *nvidia-uvm*[28]), though efficient for applications that only marginally exceed the GPU memory capacity and/or for applications with regular memory access patterns, are woefully inadequate to handle the working set needs of applications with irregular access patterns that oversubscribe to GPU memory capacity. Based on several insightful observations, we derive a significantly better paging mechanism for UVM under such over-subscribed scenarios, implement this on an existing

NVIDIA UVM solution, and experimentally demonstrate a 34% speedup on the average over the current *nvidia-uvm.*

Memory Oversubscription: Accommodating datasets larger than the available physical memory capacity, is an age-old problem that Virtual Memory has to deal with. While this has been extensively studied for host memory-disk paging optimizations, this problem when dealing with GPU-CPU memories is relatively new. Till recently, applications and/or language runtimes had to deal with the limited GPU memory capacity by themselves - either run kernels which fit and/or move data explicitly. However, UVM is expected to ease this programming burden to blur the gap between CPU and GPU memories going forward. However, current UVM implementations use either a simple demand paging mechanism, or an access-counter based lazy mechanism (i.e. bring in only after 256 accesses to a group of 16 pages) to avoid evicting (and thrashing) some other more useful pages resident in GPU memory. As we will show, when applications considerably over-subscribe to the GPU memory , the current UVM implementations result in excessive thrashing of pages and very poor performance.

Memory over-subscription will continue to be, if not become even more, important. In fact, even though the cost of memory continues to decline, the increase in memory capacity of GPUs (2.5× over the last four years) still cannot satisfy the size of datasets (4× or even larger) [9, 12, 24]. Cost differences will allow easier scaling and deployment of conventional DDR capacities on the host, compared to more specialized high-bandwidth memories on the GPU. Application sizes are also expected to increase drastically going forward, requiring processing of huge datasets that will spill over to CPU memories (and possibly even storage devices). For this reason, many prior works [9, 12, 15, 27, 32] has also used over-subscribed scenarios for large application motivations. This work specifically targets over-subscribed GPU memory executions to provide a new runtime paging environment for such scenarios, and is not intended to be used for normal/under-subscribed executions.

Irregular Applications: Dealing with memory over-subscription is relatively easier if applications have well defined/regular access patterns. At the application level (or in the compiler) itself, one could (statically) better pack the data, and tile the computations for spatial and temporal locality. Further, in the runtime, regularity and predictability can be used to prefetch data ahead of need, figure out better eviction algorithms, etc. As we will show empirically, prefetching in UVM does do well, even for over-subscribed scenarios, in regular applications. However, irregular applications are not readily amenable to such static and dynamic enhancements. In fact, prefetching can worsen performance because of the poor predictability. There are several important irregular applications (e.g. graph applications in epidemiology [7], social networks [14], web-search/internet [25], bioinformatics [2], etc.) [3, 6, 31], while are well suited to GPUs for their parallelism but have huge datasets that may not fit in the GPU memory, and are also randomly/irregularly accessed. Such applications are the intended target of this work.

Differences from Conventional Paging: While the problem seems quite similar to conventional paging between main memory and the storage device, there are some notable differences. There are obvious differences in latency and bandwidth of the components at hand, GPU-CPU memory vs. memory-disk, to require fine tuning some of the mechanisms. More important, the 2 notable differences that we will exploit include (i) Current GPUs allow their cores to directly reference and access locations on host memory through the UVM system, without requiring those pages to be brought into the GPU memory. In conventional paging, data is necessarily brought to the memory from the disk before the CPU access; (ii) Current GPUs also allow caching of the data (close to the cores) brought from host memory, without needing to adhere to the "inclusion property", i.e. what is in cache does not need to be in GPU memory [23]. Based on these two features, we can differently use the storage structures on the GPU. Specifically, we will use the GPU caches to hold data that captures temporal locality and the GPU memory to hold data that captures spatial locality, rather than duplicate both functionalities in both these levels of the hierarchy (thereby saving precious space as well) for over-subscribed scenarios. To our knowledge, there has been no such differential exploitation, for paging in GPUs.

Contributions: Targeting irregular applications running on over-subscribed GPU memory systems, this paper makes the following important contributions:

- We note limitations in current UVM implementations (*nvidia-uvm*), which incurs excessive page faults, with pages evicted (early) before they are fully utilized. Prefetching worsens this problem for irregular applications.
- Based on these limitations, we identify execution characteristics to better use the limited storage structures on the GPU. It is better to track locality at finer granularities. Further, if we examine the temporal (using LRU stack) and spatial locality of the data accesses, the former can be fulfilled with the GPU caches themselves, and is better to devote the GPU memory for spatial locality instead. This separation of responsibilities is a novel contribution of our work, especially for GPUs.
- Consequently, we focus on identifying, predicting, and exploiting the spatial locality within pages for dynamic page migration. We show that prior spatial utilization of a page, before its eviction, is a good indicator of the future. To

track this utilization, without a hardware access control/-tracking mechanism at that granularity in today's systems, we have implemented a compiler pass to instrument accesses. The runtime system uses these logs to estimate utilization, and dynamically sets an adaptive threshold (based on page fault rates) to determine migration. This decision making is based on an analytic formulation of different static and runtime parameters.

- We have implemented this compiler pass (on LLVM) and runtime system (on *nvidia-uvm*) for the NVIDIA Turing platform, into a system called DynaMap, which can readily run off-the-shelf CUDA UVM applications.
- Experimentation on an actual system, with 7 irregular applications from different public suites, shows that DynaMap provides on average, a speedup of 197% over the default NVIDIA driver and 34% over the enhanced hardware access counter based solution by NVIDIA.

2 UVM Implementation in NVIDIA GPUs

UVM creates a single address space across the DDR memory on the host CPUs and the GDDR/HBM memory on the GPUs [20] for seamless data movement (via the PCI-e bus) without requiring programmer intervention. The core runtime responsible for UVM is *nvidia-uvm*, a kernel module that is open source as part of the NVIDIA GPU driver. Programmers allocate memory regions on UVM via the *cudaMallocManaged* CUDA API. UVM then demand pages the memory allocated within these regions to the device where they are accessed (CPU or the GPU).

When the GPU accesses a non-resident location (cache and memory), a page fault is raised and the driver interrupt service routine (ISR) is invoked. The ISR performs the following: (i) Fetches the faulting address from a hardware buffer where concurrent faults from different threads are 'batched' [16] together. (ii) Moves the page group corresponding to these faulting pages using DMA engines [21][29] to the GPU based on the migration policy. The migration policy comprises (a) system enforced policies [28], (b) prefetching using the widely acknowledged [10] tree-based prefetcher and, (c) user specified hints [23]. (iii) Evicts a group of pages (LRU based), if memory space is unavailable on the GPU for the new page. Note that this LRU list is ordered based on time of allocation and *not* updated at time of access.

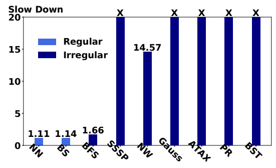
We refer to the above default demand paging system as Eager. NVIDIA also supports a less eager mechanism, where hardware counters are used to track access counts at 64KB (16 pages) page group granularity, with a static threshold (of 256) used to trigger migration (we refer to it as LazyGrpHW). A recent work [11] extends this to use page granularity counters with adaptive thresholds. We will evaluate this scheme (LazyPgDynSW) with a software implementation, as it is not realizable in current hardware.

3 UVM under memory over-subscription

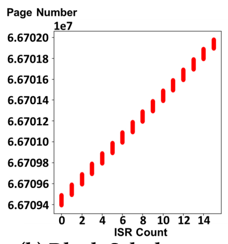
To illustrate the deficiencies of UVM under over-subscription, we conduct experiments on a representative set of applications by over-subscribing the GPU memory (see section 6 for setup). We set a nominal 100% over-subscription ratio[1] by allocating extra data in the GPU driver, so that availability to each application decreases commensurately. As Figure 1a shows, we see considerable slowdowns, $> 100\times$ in several applications because of limited memory capacity on the GPU. The primary reason for this performance degradation is the (expected) thrashing of pages between the host DRAM and the GPU's DRAM. This is further corroborated by examining their memory access patterns. While applications with regular patterns (Figure 1b), suffer minor slowdowns, irregular access pattern (Figure 1c) applications exhibit worse spatial locality (distinct pages are accessed in a short time) and temporal locality (high temporal separation for faults to the same page) and incur a high degree of thrashing.

There are two more factors which can worsen the performance, ineffective prefetching and premature eviction. Prefetching may be harmful for irregular applications under over-subscribed scenarios. As shown in Figure 1d, the performance of PageRank improves by 41% when we disable prefetching. Not only are many of the fetched pages not very useful, they occupy precious space evicting some other more useful pages. This becomes challenging when dealing with irregular applications where the what/when to prefetch question is a lot harder to answer. Besides, premature eviction can exacerbate thrashing. Even though prior research [10, 16, 17] has proposed efficient ways for page evictions, large working sets, coupled with large granularity, can cause less useful in-demand pages to evict more useful page in the over-subscribed GPU memory. We see evidence in Figure 1e where the evictions normalized to number of references increase with over-subscription ratio. Also, we plot the frequency of references to the distances in the LRU stack below the memory capacity limit, i.e. these would be faults because the ones on the top of the stack will be in memory for LRU, in Figure 1e. we plot the CDF of this for different over-subscription ratios. As we see, the over-subscription skews the curves to the left as the over-subscription ratio increases. This clearly shows evidence of earlier eviction of those units than what the optimal eviction (the evicted units should percolate deep down in the stack) would have done by trying to retain those units longer in memory (before eviction). In summary, while UVM is adequate for cases where there is sufficient GPU memory, there is a severe performance penalty under over-subscription, especially when application references are sparse and irregular.

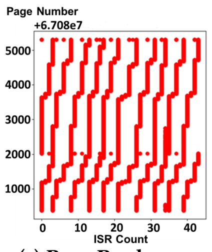
[1]Over-subscription ratio is defined as (problem size - memory capacity) / memory capacity.



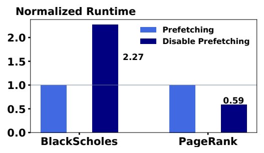
(a) Performance degradation with 100% oversubscription



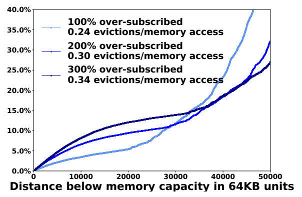
(b) Black Scholes (Regular)



(c) Page Rank (Irregular)



(d) Prefetching disabled with 100% over-subscription ratio



(e) CDF of the number of unique 64KB referenced since evictions

Figure 1: UVM under over-subcription

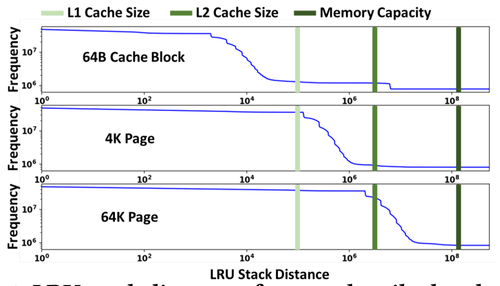


Figure 2: LRU stack distance of over-subscribed scaled-down Gaussian application for different granularities

4 Requirements for Over-subscribed Scenarios

As observed, UVM brings in pages eagerly, thereby evicting pages that are not fully utilized, and causing severe thrashing under over-subscription. One way to address this "eagerness" problem is to delay the migration [11], wherein pages can be *soft-pinned* on the host memory, and the GPU accesses for these pages can be tracked by access counters, (since NVIDIA Volta [28]), with a threshold which when hit automatically migrates the pages to the GPU. However, we have seen that the counters are at a 64 KB granularity, rather than for individual page level tracking, and the threshold is pre-determined (set to 256) in the NVIDIA driver. While arresting the eagerness, such an approach still does not address many requirements. As we will show: (i) A granularity of 64KB for tracking these accesses is too coarse and misses out on the utilization of individual pages when determining when/whether to migrate. (ii) Access counters (even at the page level) do not distinguish between accesses to different parts of a page, i.e. the same cache block of a page could be referenced multiple times (temporal locality) and the access counts cannot differentiate those from accesses to different parts (spatial locality) of a page. (iii) The eagerness factor for migrating pages should depend dynamically on the GPU memory pressure. We illustrate these deficiencies below, and use these results to derive requirements for a better strategy.

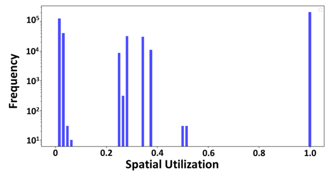
4.1 Granularity of page tracking

To amortize the cost of tracking and leverage efficiencies of bulk data transfer, current page management schemes (hardware counter based) and prefetching mechanisms in NVIDIA drivers track accesses at a page or higher granularity. This is based on a reasonable expectation that neighboring pages are likely to be accessed in the near future. While this may be true for regular memory access applications that sweep data structures, the same cannot be said of irregular applications that are quite sparse and random in accessing their data structures. To illustrate this, we study the LRU stack distance [4] (a measure of temporal locality) of pages accessed by an irregular application with different page granularities.

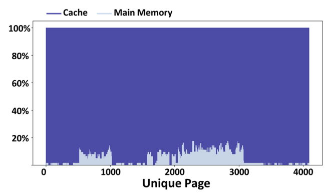
As seen in Figure 2 for the same irregular application (Gaussian [26]), the LRU stack distance plots shift considerably to the right (i.e. worse temporal locality) as the granularity of tracking increases from a cache block (64 B) to a 4K page and a 16 page (64 KB) granularity. This can be explained as follows. If the application accesses every byte of the corresponding granularity in a spatially contiguous fashion, the stack distance would likely be similar across the granularities. However, when *not every byte is accessed* - especially as the granularity gets large (e.g. 64 KB) and the access pattern is sparse, the LRU stack distance gets larger because there would be addresses/bytes in the stack that are closer to the top of the stack (since they fall within the same unit as some other byte in the same unit that is recently referenced) than the currently referenced address, This problem accentuates with the sparsity of references (irregular accesses) and the coarser granularity of tracking.

The 3 vertical lines of the figure correspond to L1, L2, and memory capacity of the GPU. We see that at the cache line granularity, the first working set (the first knee where the curve drops steeply) occurs at a stack distance which can fit in the size of the GPU L1 cache. With the page granularity, it takes up to L2 capacity to fit this working set. However, with 64K granularity, neither L1 nor L2 can fit the working set even if they were to use a fully associative LRU cache.

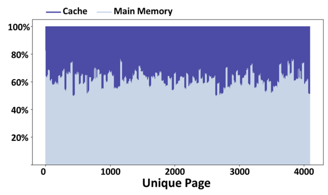
These results point to the need for a finer granularity of (i) tracking accesses to determine what we want to retain and what we want to evict, and (ii) determining the specificity of bringing in data from the host to the GPU to avoid bringing in "non-referenced" data. While we would like to go as low as a cache block (64B), the problems are (i) address translation at that granularity cannot be supported in hardware (due to



(a) Distribution of spatial utilization of pages migrated to GPU memory before eviction



(b) Location of service for low-utilized pages



(c) Location of service for high-utilized pages

Figure 3: Spatial utilization of pages for the Gaussian application with 200% over-subscription

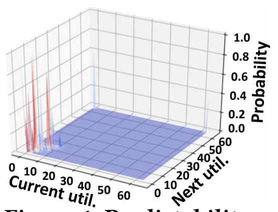
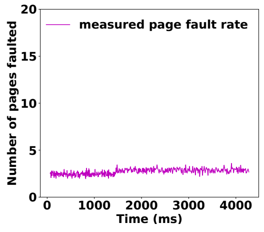
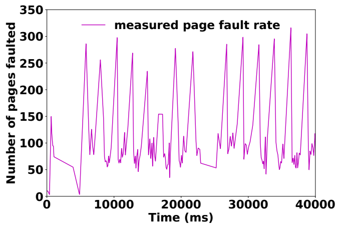


Figure 4: Predictability of spatial utilization



(a) NN (Regular)



(b) PageRank (Irregular)

Figure 5: Measured page fault rate

high overheads), and (ii) becoming more latency constrained (not exploiting higher bulk data transfer bandwidths) for moving units between host-GPU. Our solution, described later, will still *track accesses at cache block granularities*, so as to be very discretionary about what to bring in and evict, but at the same time *transfer and map at page granularities* when migration is needed, providing a good trade-off between the cache block and 64KB extremes.

4.2 Capturing spatio-temporal access characteristics

While hardware access counters do provide a direct measure of estimating temporal locality, they only provide an indirect way of inferring spatial locality, i.e. right shift of the higher granularities in Figure 2 is a consequence of poor spatial locality. We corroborate this observation of spatial locality for the same Gaussian application in Figure 3a which shows the distribution of spatial utilization of pages migrated to the GPU before their eviction. As seen, there is a large number of pages with very low spatial utilization, but are nevertheless migrated to the GPU. In fact, these are just a couple of cache lines within a page, which would automatically get cached in the GPU's caches, and do not even need to be present in GPU memory[2] This is clearly the case when we examine where requests to these pages are serviced from in Figures 3b and 3c. We see that on average 91% of the requests for low utilized pages are serviced from GPU caches (Figure 3b) while for high utilization pages they are serviced equally from both the GPU memory and caches (Figure 3c). This observation suggests a very different way of deciding what pages to migrate to GPU memory, and how to service the remaining pages: migrate pages where a large number of its cache blocks will be utilized (i.e. with good spatial locality), and leave pages with low utilization in the host memory without migration and rely on the GPU caches to exploit the temporal locality to service those requests. In other words, we should *use the GPU caches for leveraging the temporal locality in the application (and not put the corresponding pages in the GPU memory), while we will use the GPU memory for those pages with high spatial locality.* This is a novel way of using the different layers of the memory hierarchy to optimize for different kinds of locality, as opposed to optimizing for both spatial and temporal locality in each layer. The GPU caches automatically identify and retain (using LRU) high locality data in the temporal dimension, and we only need to identify and retain (in GPU memory) high spatial locality/utilization data.

[2]Note that NVIDIA GPUs allow data from host memory to get cached without requiring the corresponding pages to be brought into its memory [23].

4.3 Predictability of Spatial Utilization

A natural question next is how to estimate the spatial utilization of a page, especially for sparse and irregular applications. To answer this, we measure the spatial utilization of a page when it is brought into the GPU memory (current) and the time when it is brought in after an eviction (next) and then calculate the probability of this correlation. We observe that the current and next utilization for a page during its residence in the GPU are highly correlated as shown in Figure 4. This suggests that *if we could somehow track the utilization of a page when it is brought into memory the first time, even if it was a bad decision, we could use this information for the future to avoid bringing it in if the value is low.*

4.4 Dynamically adapting to memory pressure

While the above arguments help guide the eagerness factor in controlling when and whether to migrate a page based on the spatial and temporal locality, it is equally important to take into account the memory pressure at that point in time. Applications can go through widely varying phases, with corresponding page fault rates widely different, as Figure 5b shows. Consequently, the cost vs. utility of migrating a page to the GPU memory also varies across these phases. When the page fault rate is high (higher memory pressure), we would like to be more aggressive in determining the utility of a page as opposed to when page faults are rare. Similarly, proactive page management actions like prefetching need to be more careful and/or even turned off when memory pressure is high. Employing a static/single threshold value (i) across all applications (may suffice for NN as depicted in Figure 5a but not for PageRank) and (ii) for their entire execution, to determine the eagerness factor for migration to GPU memory, can be highly sub-optimal.

5 DynaMap

We use the following guidelines in building our UVM system for over-subscription: (i) We should not be too eager in migrating pages on the first reference; (ii) the migration should be done for more spatially useful/utilized pages; (iii) specifically even if a small part of a page is reused multiple times (temporal locality) we should simply fetch those cache lines into the GPU cache rather than the page into the GPU memory, and reserve GPU memory for those pages with higher distinct cache lines referenced (spatial locality); (iv) this utilization should be tracked at a page granularity rather than at a macro (64 KB in NVIDIA's hardware solution) level; (v) we can reuse prior utilization history of a page to predict future utilization; and (vi) a static threshold of this utilization will not suffice to determine the migration since there is dynamic variability in memory pressure.

We next design and develop a complete UVM runtime system, called DynaMap, using these guidelines to be readily deployed on current NVIDIA GPUs, to accommodate applications out-of-the-box without any programmer intervention. It should also be easy to port to other GPUs. Our system has 3 main parts: (i) a compiler pass over the application sources to insert instrumentation code to track spatial locality within pages, (ii) a cost model that dynamically determines threshold for page utilization to trigger migration, and (iii) a runtime modification to the NVIDIA device driver that gathers page utilization (from (i)) to predict future utilization, and dynamic thresholds (from (ii)), and accordingly decides whether or not to migrate each page upon a fault.

5.1 Compiler-inserted instrumentation to track utilization

To calculate the spatial utilization for each page during execution, we need to track the addresses accessed by each load or store - at the cache line granularity. However, the hardware itself tracks accesses at only higher granularities (in NVIDIA [28] this is done at 64KB or 16 page granularity). Even on host CPUs, accesses (using reference bits in the page table) are tracked at only a page granularity, not providing utilization within the page. Setting permissions to trap on every page access to track cache lines can lead to considerable overheads - it takes roughly 20μs to simply invoke and return from a page fault handler with no data transfer.

Instead, we employ an instrumentation approach to perform access tracking at a cache line granularity. This instrumentation has to be extremely efficient so as to not introduce significant overheads that defeat its purpose. We use the compiler for the GPU to insert highly optimized instrumentation code for this purpose. We add a LLVM middle-end [19] pass to patch instrumentation code after each load and store instruction to track the addresses that are accessed. This code stores the information onto a set of data-structures that we call as the page statistics table (PST), which is itself allocated and pinned to the host memory. The GPU runtime system is informed of the address of the PST at the start of program execution. The PST is composed of two tables. The first table tracks data structure level allocations as defined by the programmer. It contains the *start-address, end-address, size, cache-table-ptr* of each data region. Each row in this table points (*cache-table-ptr*) to a secondary table that tracks the usage of each cache line allocated within every page accessed by this data region. The secondary table is simply a list of elements (one per cache line) counting their usage. Size of each row in the primary table is 32 bytes, and the size of each row in the secondary table is 1 byte. After each load/store operation, the inserted instrumentation code first looks up the first level table based on the data region accessed and determines the pointer to the secondary table. It then sets the corresponding entry byte of the cache line in the secondary table. The runtime system can subsequently examine this secondary table to determine the utilization of each page before deciding its placement. As observed earlier, if we have done this before, we have good predictability of future utilization, to avoid the re-tracking of these accesses.
Optimizations: In order to minimize the instrumentation overheads, we have implemented several optimizations:

- *pin-to-cpu:* Pinning and allocating the PST on the CPU memory allows precious GPU memory to be completely used by the application to not worsen the over-subscription.
- *read-and-bypass:* Since writes are costlier than reads, rather than just overwriting (setting) the secondary table byte upon an access of the corresponding cache line, we first check if it is already set and, if so, we avoid the subsequent write.
- *early-stop:* Once the runtime has determined where a certain page has to be mapped, it sets a flag for the instrumentation code to skip tracking for those pages. This avoids incurring the tracking costs for those pages.
- *batch update:* Multiple updates to the PST can be batched together to reduce the number of operations and thereby amortize the costs. At the instruction level the compiler can identify load and store instructions that access the same memory address to batch those updates in the PST,

eliminating some of the redundant updates to those entries [8]. Further, by exploiting conventional latency hiding techniques of the GPU, updates to the PST can also be batched at the warp/block level. CUDA kernels are usually written in a form where global memory is accessed based on offsets of block and thread indices. The compiler can identify such access patterns and tweak the instrumentation code to make a single warp / block / thread perform the update for a bunch of neighboring warps / blocks / threads. Other warps need not be blocked consequently, and can continue their execution meanwhile. A decrease in accuracy is expected by batching in-flight instructions / warps that are not yet complete. To minimize this, we find that a small batch size of 4 warps provides good performance outweighing its deficiencies.

The instrumentation costs can still be considerable, 40 PTX instructions and 10 LD/ST instructions for a single update to the PST and taking around 6000 cycles, but note that (i) they will be incurred only in over-subscribed scenarios and (ii) the benefits can overshadow these overheads in such situations as will be shown in our evaluation. The size of PST is D/L bytes (D: data-set size, L: cache line size) and is always allocated on the host memory. Allocations range up to 192 MB for the applications that we evaluated.

5.2 Light-weight runtime system

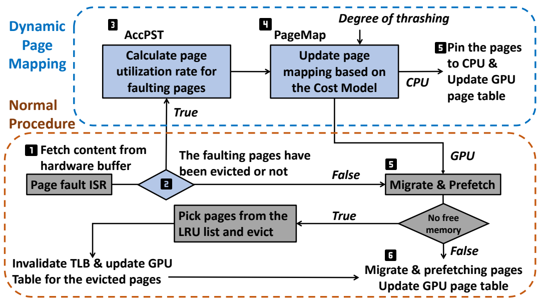

Dynamic Page Mapping
Normal Procedure
3 AccPST
Calculate page utilization rate for faulting pages
4 PageMap
Degree of thrashing
Update page mapping based on the Cost Model
CPU
5 Pin the pages to CPU & Update GPU page table
True
1 Fetch content from hardware buffer
Page fault ISR
2
The faulting pages have been evicted or not
False
GPU
5
Migrate & Prefetch
Pick pages from the LRU list and evict
True
No free memory
False
Invalidate TLB & update GPU Table for the evicted pages
6
Migrate & prefetching pages Update GPU page table

Figure 6: Enhancements in *nvidia-uvm* for DynaMap

There are 2 main functionalities in this system: (i) logging and collecting the page access statistics recorded by the instrumentation code, and (ii) using the statistics to perform adaptive page migration/replacement. We have implemented this in the kernel module called *nvidia-uvm* as part of the NVIDIA GPU driver (v440.33). The basic flow of handling a page fault inside *nvidia-uvm* is as shown in Figure 6 (highlighted by a red-box). When a page-fault is raised on a page group (2MB granularity), the ISR for the page fault migrates and prefetches neighboring pages. If there is no free memory available, the runtime system picks the LRU page group and evicts pages that are present in the GPU memory. More details regarding this can be found in the uvm_va_block_service_locked function in the driver.

Two interfaces are added to this kernel module (shown in blue) - (i) *AccPST*: to access the PST, and (ii) *PageMap*: to update page mapping. The *AccPST* interface looks up the PST and calculates the spatial utilization of a page. The *PageMap* interface updates the page mappings of any page as determined by the dynamic thresholds described next.

5.3 Adaptive Eagerness Threshold

The decision to migrate a page (to GPU memory) must also be cognizant of the dynamic pressure experienced by the memory system due to over-subscription. We use the dynamic page-fault rate ($f_{current}$), at any time, as tracked by the device driver as a direct measure of this memory pressure/over-subscription. We need to weigh the costs of leaving a page on the host memory vs. the costs of bringing in that page to the GPU memory in order to determine whether to migrate the page to the latter. The table below defines the terms and the corresponding values on the experimental platform, that we will use in the discussion.

Variable	Description	Value
x	Number of bytes accessed	0-4096 bytes
L	Cache line size in bytes	128 bytes
f_{over}	Expected number of evictions	eq.(2)
P	Page size	4096 bytes
B_C	Bandwidth of CPU memory	12.8 GB/s
B_G	Bandwidth of GPU memory	336 GB/s
B_{PCI}	Bandwidth of PCI-e bus	16 GB/s
H	Page fault handling time	20 us

The cost of accessing x bytes in a "faulting" page that is left on the host memory (and not brought into the GPU) can be expressed as:

$$C_{CPU} = \frac{\lceil x/L \rceil * L}{B_C} + \frac{\lceil x/L \rceil * L}{B_{PCI}} + \lceil \frac{x}{L} \rceil * H \qquad (1)$$

The first 2 terms represent the costs of reading the x bytes from the host memory and transferring them over the PCI bus respectively. The third term represents the cost of page fault handling H, where $\frac{x}{L}$ represents the number of faults that would be incurred to this page (since we are bringing cache lines at a time into the GPU caches).

On the other hand, if the entire page was brought into the GPU memory, the cost of accessing those bytes would have taken $H + \frac{x}{B_G} + \frac{P}{B_C} + \frac{P}{B_{PCI}}$, barring other issues. The last two terms are the cost of moving the entire P bytes of the page to the GPU, the second term representing the cost of accessing x bytes from GPU memory, and H denotes the single page fault cost. However, this would be the case only if there is no eviction of another useful page in GPU memory (premature eviction) caused by this migration. Early on, when the application starts on the GPU, its page faults will not cause any evictions from its memory. Let us denote this as f_{under}, i.e. page fault rate before evictions start. We would like page fault rate at any subsequent time, $f_{current}$ to be close to f_{under}, implying that we are not incurring excessive capacity-related faults (i.e. evictions are not significant). However, over-subscription to GPU memory can cause evictions to increase, and this increase at any time (f_{over}), relative to the base page fault rate (f_{under}), can be estimated as follows.

$$f_{over} = (f_{current} - f_{under})/f_{under} \quad (2)$$

f_{over} is thus an estimation of the number of additional faults/migrations due to premature eviction induced by demand migrating the currently referenced page from the CPU to the GPU. Hence, such a migration effectively results in $1 + f_{over}$ faults, and the total effective cost of migrating that page to the GPU can be calculated as:

$$C_{GPU} = \frac{x}{B_G} + (1 + f_{over}) * [H + P * (\frac{1}{B_C} + \frac{1}{B_{PCI}})] \quad (3)$$

To dynamically determine whether to migrate the faulting page or not, we simply chose the option that has the lower cost (i.e. C_{CPU} or C_{GPU}). The importance of tracking and accounting for all these factors is depicted in Figure 7 which plots C_{CPU} and C_{GPU} as a function of spatial utilization x, for different levels of over-subscription.

As can be seen, when there is no thrashing/over-subscription, it makes sense to be more aggressive towards migrating the page from the CPU to the GPU. However, as over-subscription levels increase (f_{over}), the cross-over point of costs shifts more to the right. This clearly shows the need to track these dynamically and accordingly make the migration decision.

On a page fault to a page that has already been evicted before, we determine the spatial usage (in bytes) x for that faulting page using the *AccPST* interface. In addition, the driver continuously measures the page fault rate ($f_{current}$) and calculates the f_{over} metric to ascertain the current pressure on the memory system. It can then calculate the two costs, C_{CPU} and C_{GPU}, and decide whether to migrate the page.

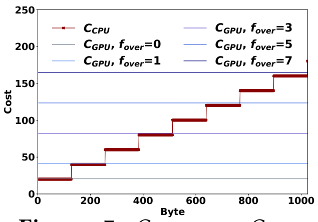


Figure 7: C_{CPU} vs. C_{GPU} costs for x bytes accessed for different f_{over} rates.

6 Evaluation

Application	Description
BFS (G)	Breadth First Search
SSSP (G)	Single-Source Shortest Path
NW (R)	Needleman-Wunsch, DNA sequence alignment
Gaussian (R)	Gaussian Elimination
ATAX (P)	Matrix Transpose and Vector Multiplication
PageRank (H)	Websites ranking algorithm
BST (H)	Binary Search Tree

Table 1: Benchmark (G: graphBIG [22], R: Rodinia [5], P: PolyBench [13], H: HeteroMark [30])

	GPU	CPU
	GeForce RTX 2060	Intel Core i7
Cores	30 SMs, 64 CUDA cores/SM	4 cores, 2 threads/core
Clock	Max clock 1680 MHz	3.4 GHz
Memory	5935 MB GDDR6, 336 GB/s	32 GB DDR3, 12.8 GB/s
Cache	L2: 3 MB, L1: 96 KB/4 Blocks	32/256/8192 KB
Bus	PCIe 3.0 x16, 16 GB/s	

Table 2: System Configurations

6.1 Experimental Setup

Workloads: We evaluate our proposal, together with current approaches, on some well-known irregular applications as listed in Table 1.

Platform: Table 2 describes the actual GPU platform running Ubuntu 18.04.4 (Linux 4.15.0) on which our evaluations are performed. The default *nvidia-uvm* kernel module is replaced by the modified one running DynaMap.

Methodology: We target off-the-shelf CUDA applications and modify them to use `cudaMallocManaged`. We compile these applications using LLVM to incorporate the instrumentation code for tracking spatial usage. Since our goal is to reduce thrashing under over-subscription, unless specified otherwise, we set the memory over-subscription ratio at 200%. We also vary over-subscription ratios.

Schemes: We evaluate the following:

- `Eager` is the default on-demand paging system inside NVIDIA UVM with prefetching enabled.
- `Eager-woPref` is the default on-demand paging system inside UVM with prefetching disabled.
- `LazyGrpHW` is the state-of-the-art on NVIDIA GPUs. This uses hardware access counters to track pages at 64KB granularity (16 pages) and migrates pages when the count goes beyond a fixed threshold value of 256.
- `PinHost` wherein all the allocated data is pinned to the CPU memory using `cudaMemAdvise`.
- `LazyPgStatSW` is a policy similar to `LazyGrpHW` with the tracking and migration done for individual pages at 4K granularity. We fix the threshold value statically at the best performing value for each application after varying them for 2, 8, 32, 64, 256. This is emulated in software since page level tracking is lacking in hardware today.
- `LazyPgDynSW` is the policy proposed in [11] with granularity of page tracking at 4KB and migration based on dynamic thresholds as determined by equation (1) in [11]. This is also emulated in software.
- `StaticMap` is a static policy that tracks spatial usage and migrate pages with a specified threshold (=0.5).
- `DynaMap` is our proposed solution which (i) tracks page accesses at cache block granularity, and (ii) dynamically adapts to memory pressure as described in Section 5 to perform page level migration.

6.2 Comparing the Schemes

Figure 8 shows the speed up of the applications for the considered schemes, over `LazyGrpHW`.

Improvement over Baselines: The possible mechanisms in today's systems are `Eager`, `Eager-woPref` and the more recent `LazyGrpHW`. Of these, due to the thrashing effects we observed earlier in section 3, the `Eager` and the `Eager-woPref` policies perform the worst of the lot as they eagerly migrate pages, leading to early evictions, without being aware of

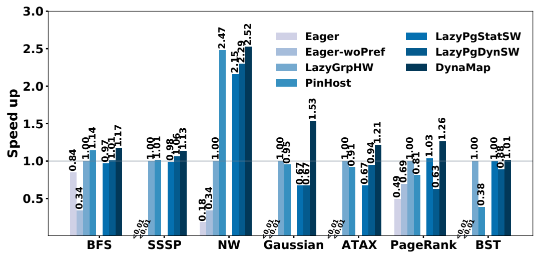


Figure 8: Normalized Speedup of UVM applications with 200% over-subscription ratio (normalized to LazyGrpHW)

their spatial utility for the GPU. In fact, in a few applications, these two policies do not even run to completion - we mark such cases by <0.01 in Figure 8. On average, DynaMap outperforms Eager by 197% across all the applications which run to completion.

To better understand the performance of DynaMap over the state-of-the-art LazyGrpHW, we compare the rate of migrations/evictions of pages in these two schemes for the PageRank application in Figure 9 during its execution Recall that there are two key differences between these two schemes. First, the granularity of tracking in *LazyGrpHW* is at a coarse 64KB granularity, while in our scheme we track pages at finer 4KB sizes. Second, *LazyGrpHW* tracks the temporal use of the 64KB unit and migrates it when it goes beyond a threshold - regardless of cache blocks accessed (possibly could be a small number that are repeatedly accessed) - as opposed to our scheme where we migrate based on the spatial usage of a page. As we can see, under *LazyGrpHW*, once the GPU memory is oversubscribed, there is frequent migration of pages into the GPU. Not only are units being constantly fetched, these fetched units are much larger (64KB) as opposed to our *DynaMap* policy where small 4KB pages are trickling in based on their predicted spatial use. Apart from the cost of migrating much larger volumes of data, such migration also leads to more frequent early evictions. Upon measuring the average number of evictions per page, we see that under LazyGrpHW every page is evicted 508 times, as opposed to DynaMap where they are evicted only 21 times, making DynaMap achieve a speedup that is on average 34% higher across the 7 applications.

Spatial utilization vs. Access counts Thrashing in LazyGrpHW may be due to the large tracking granularity or due to having a static threshold for access counts. Digging deeper, we compare it against LazyPgStatSW where the tracking of pages is at 4KB while still using access count thresholds. Comparing the two, we see no clear winner - LazyPgStatSW does better than LazyGrpHW in NW, but much worse in Gaussian, and comparable in others. More importantly, in Figure 10 compared to DynaMap, the page fault rate of LazyPgStatSW is higher, indicating significant page migrations/evictions even with fine-grained tracking of pages. One may then wonder whether a dynamic threshold for the count would

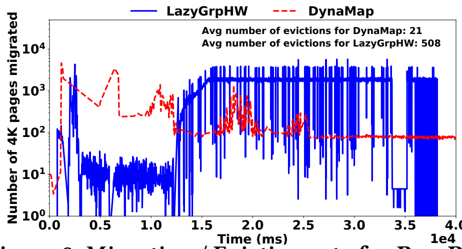


Figure 9: Migration / Eviction rate for PageRank

provide the same benefits as DynaMap. We have implemented LazyPgDynSW from a recent work [11] where page migration is based on a dynamic threshold of access counts. Though this does somewhat better than choosing static thresholds, DynaMap still performs significantly better, with a speedup that is 37% higher (compare page fault rates in Figure 10).

This clearly shows that under over-subscription, (i) page level policies are better than those with higher granularities, and (ii) solutions that are based on temporal locality have considerable thrashing even with dynamic thresholds, justifying our motivation to track and migrate based on spatial usage of pages.

Better than pinning always on host: One may wonder if the over-subscription is so bad, then we may have as well left all pages on the host memory, never migrating them. Such an approach, denoted as PinHost, does significantly better than the 2 always migrate baselines considered above (Eager and Eager-woPref). It is also comparable to the LazyGrpHW mechanism, doing better than the latter in some cases and vice-versa in others. For a few applications (BFS, SSSP), PinHost comes quite close in performance to our DynaMap. However, on the average, DynaMap still beats PinHost by about 21%. The performance variance across the applications for this behavior can be explained by the differences in the sparsity of accesses across these application. In BFS and SSSP, as opposed to applications like PageRank and BST, most of the pages have low spatial use. Consequently, it is not a bad choice to keep all the pages on the CPU memory. In fact for BFS and SSSP the DynaMap policy also chooses to keep most of the pages on the CPU memory. However, for other applications, keeping all pages on the CPU can be more expensive as there are quite a few pages with significantly higher spatial usage. This clearly shows the value of allowing pages with high spatial use to be brought into the faster GPU memory.

The above results reaffirm our view that we need to be defensive in allowing pages to flow into GPU memory especially when it starts to be over-subscribed. Instead of having policies that allow all pages to come into the GPU or pins all pages in the CPU memory, our adaptive strategy brings in pages with good spatial utility.

Adaptive vs. Static utilization threshold: While DynaMap uses a dynamic spatial utilization threshold, a natural question is whether a static threshold value would suffice. To demonstrate the need for an adaptive threshold, we

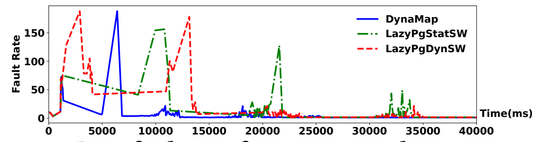


Figure 10: Page fault rate for DynaMap and LazyPgStatSW

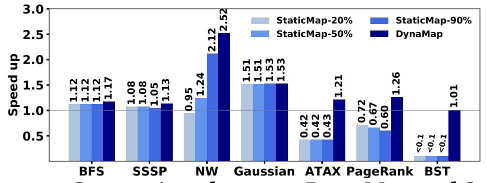


Figure 11: Comparison between DynaMap and StaticMap

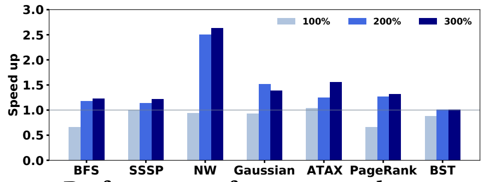


Figure 12: Performance of DynaMap under 100%, 200% and 300% over-subscription (normalized to LazyGrpHW)

compare DynaMap to a set of static policies (StaticMap) parameterized by predetermined threshold values of 20%, 50%, and 90% of spatial utilization of a page. As seen in Figure 11, the overall performance of the static and dynamic policies are very similar for some applications like BFS, SSSP, and Gaussian. This is because, for these applications, most of the evicted pages have low spatial utilization and thus all of them are prevented from coming to the GPU whether it be dynamic or static thresholds. However, for PageRank, StaticMap performs 76%, 89%, and 110% worse than DynaMap for the 3 different threshold values respectively. The performance worsens as the value of the threshold increases, implying we should be less rigid about migrating a page to the GPU. On the other hand, for NW, StaticMap performance is 56%, 51%, and 41% worse than DynaMap, implying we should be more rigid in migration as the early eviction problem in this application is worse.

Over-subscription ratios and software-only limitations: As shown in Figure 12, under lighter over-subscriptions (100%), except for ATAX, access counter based LazyGrpHW shows better performance than DynaMap (16% on average). Under such conditions, the profiling overheads outweigh any performance gains from the page placement. However, smarter page placement outweighs the overheads under moderate to heavy over-subscription ratios, providing 34% and 41% (on average) improvements in speedup for 200% and 300% over-subscription respectively. To obviate such overheads, we can profile the application separately and use it to perform page mapping in a subsequent execution. **In such cases, DynaMap improves performance further by 38% on average (for 200% over-subscription**).

6.3 Summary

DynaMap establishes the need to track sub-page spatial usage and set thresholds dynamically in performing the page migration. The results clearly suggest the need for exploiting spatial utilization within a page, and dynamically setting a threshold for this based on current memory pressure, to determine page migration as done in DynaMap. DynaMap is implemented in software only because current hardware does not expose the facility to track fine-grained accesses, making a compelling case for incorporating such tracking information in future hardware, e.g. using a bit map of referenced cache blocks in page table entries/TLBs that would take 32 bits per page, which is not very expensive.

7 Related Work

Ganguly et al.[10] propose locality-and-prefetcher-aware pre-eviction policies to deal with over-subscription. ETC [17] improves GPU performance under over-subscription using proactive eviction, memory-aware throttling, and capacity compression. Li et al.[18] develop compiler-runtime collaborative strategies to adaptively choose implicit and explicit data transfer to deal with problems of unified memory. Kim et al.[16] increase the batch size to amortize fault handling time and overlaps page evictions with CPU-to-GPU migrations to improve performance under over-subscription. Nevertheless, the amortized page fault handling time is still expensive to the system and it cannot eliminate page thrashing completely. A recent work [11] proposes a dynamic threshold based policy based on access counters. We compare and evaluate against a software implementation of it (LazyPgDynSW) and show our dynamic spatial utilization based approach provides much lower thrashing behavior.

8 Concluding Remarks

DynaMap is a novel way to migrate pages for irregular applications that over-subscribe GPU memory. It is based on a unique way of differentially exploiting the temporal and spatial localities, with only the latter needing consideration for page placement. Since there is no hardware support today for tracking spatial locality within a page, we have resorted to a software only instrumentation approach, which adds considerable overheads. Though these overheads are outweighed by the benefits of better page placement in over-subscribed settings, DynaMap is not meant to be a panacea for all applications nor all over-subscription factors. This work has also suggested the need for incorporating fine grain access tracking and exposing appropriate interfaces to the software, to reduce the overheads in DynaMap, and make it applicable across more diverse scenarios.

9 Acknowledgements

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF grants 1909004, 1714389, 1912495, 1629915, 1629129, 1763681.

References

[1] Guru 3D. 2020. GDDR6 significantly more expensive than GDDR5. https://www.guru3d.com/news-story/gddr6-significantly-more-expensive-than-gddr5.html

[2] Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David Lipman. 1990. Basic Local Aligment Search Tool. *Journal of molecular biology* 215 (11 1990), 403–10. https://doi.org/10.1016/S0022-2836(05)80360-2

[3] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[4] Calin CaBcaval and David A Padua. 2003. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 150–159. https://doi.org/10.1145/782814.782836

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*.

[6] Stephen V. Cole and Jeremy Buhler. 2017. MERCATOR: A GPGPU Framework for Irregular Streaming Applications. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 727–736. https://doi.org/10.1109/HPCS.2017.111

[7] Thomson Comer. 2020. Accelerating Geographic Information Systems (GIS) Data Science with RAPIDS cuSpatial and GPUs. https://medium.com/rapids-ai/acclerating-gis-data-science-with-rapids-cuspatial-and-gpus-fd012b27af0a

[8] Chandramohan A. Thekkath Daniel J. Scales, Kourosh Gharachorloo. 1996. Shasta: A Low Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[9] Alex Fender. 2020. Tackling Large Graphs with RAPIDS cuGraph and CUDA Unified Memory on GPUs. https://medium.com/rapids-ai/tackling-large-graphs-with-rapids-cugraph-and-unified-virtual-memory-b5b69a065d4

[10] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture.*

[11] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 451–461.

[12] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1119–1133. https://doi.org/10.14778/3384345.3384358

[13] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of Innovative Parallel Computing.*

[14] Kirsten Hildrum and Philip S. Yu. 2005. Focused Community Discovery. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM '05)*. IEEE Computer Society, USA, 641–644. https://doi.org/10.1109/ICDM.2005.70

[15] G. Janssen, V. Zolotov, and T. D. Le. 2019. Large Data Flow Graphs in Limited GPU Memory. In *2019 IEEE International Conference on Big Data (Big Data)*. 1821–1830. https://doi.org/10.1109/BigData47090.2019.9006198

[16] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. "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.* 1357–1370.

[17] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. 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.*

[18] Lingda Li and Barbara Chapman. 2019. Compiler Assisted Hybrid Implicit and Explicit GPU Memory Management under Unified Address Space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 51, 16 pages. https://doi.org/10.1145/3295500.3356141

[19] LLVM. 2020. The LLVM Compiler Infrastructure. https://llvm.org/devmtg/2019-04/talks.html

[20] Inc. Micron Technology. 2019. *GDDR Memory Enabling AI and High performance Compute.* https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9968-gddr-memory-enabling-ai-and-high-performance-compute-presented-by-micron.pdf

[21] David S. Miller, Richard Henderson, and Jakub Jelinek. 2020. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

[22] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and ChingYungLin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions.. In *International Conference for High Performance Computing, Networking, Storage and Analysis.*

[23] NVIDIA. 2020. CUDA TOOKIT DOCUMENTATION. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[24] Zaid Qureshi, Vikram Sharma Mailthody, Seung Won Min, I Chung, Jinjun Xiong, and Wen-mei Hwu. 2020. Tearning Down The Memory Wall. In *Arxiv pre-print.*

[25] Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD Parallelization of Applications That Traverse Irregular Data Structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494989

[26] Adrien Rémy. 2015. Solving dense linear systems on accelerated multicore architectures. (07 2015).

[27] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/

[28] N. Sakharnykh. 2017. Unified Memory on Pascal and Volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf

[29] L. Semiconductors. 2020. Scatter-Gather DMA Controller IP. http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores01/ScatterGatherDMAController

[30] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC).*

[31] Stephen W. Timcheck and Jeremy D. Buhler. 2020. Reducing Queuing Impact in Irregular Data Streaming Applications. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 22–30. https://doi.org/10.1109/IA351965.2020.00009

[32] Tao Zhang, Jingjie Zhang, Wei Shu, Min-You Wu, and Xiaoyao Liang. 2015. Efficient Graph Computation on Hybrid CPU and GPU Systems. *J. Supercomput.* 71, 4 (April 2015), 1563–1586. https://doi.org/10.1007/s11227-015-1378-z