DREAM: Device-Driven Efficient Access to Virtual Memory

Nurlan Nazaraliyev
University of California, Riverside
Riverside, CA, USA
nnaza008@ucr.edu

Elaheh Sadredini
University of California, Riverside
Riverside, CA, USA
elahehs@ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
Riverside, CA, USA
naelag@ucr.edu

ABSTRACT

Graphics Processing Units (GPUs) excel at high-performance computing tasks, including multimedia rendering, crypto-mining, deep learning, and natural language processing, due to their massive parallelism and high memory bandwidth. However, the growing size of models and datasets in these domains increasingly exceeds the memory capacity of a single GPU, resulting in significant performance overheads. To mitigate this issue, developers are often forced to partition data and manually manage transfers between GPU and host memory—a labor-intensive approach that becomes impractical for workloads with irregular memory access patterns, such as deep learning, recommendation systems, and graph processing. Programming abstractions like Unified Virtual Memory (UVM) simplify development by offering a unified memory space across the system and handling data transfers automatically. Unfortunately, UVM introduces substantial overhead due to frequent OS involvement and inefficient data movement, particularly when GPU memory is oversubscribed. This paper presents DREAM, a GPU memory management system that leverages an RDMA-capable network device to implement a programmer-agnostic lightweight virtual memory system, eliminating CPU/OS involvement. DREAM supports on-demand page migration for GPU applications by delegating memory management and page migration tasks to GPU threads. Since current CPU architectures do not support GPU-initiated memory management, DREAM uses a network interface card to enable efficient, transparent page migration. By offloading memory management to the GPU, DREAM achieves up to 4× higher performance than UVM for latency-sensitive applications while maintaining user-friendly programming abstractions that eliminate the need for manual memory management.

ACM Reference Format:
Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. 2025. DREAM: Device-Driven Efficient Access to Virtual Memory. In *Proceedings of The 39th ACM International Conference on Supercomputing ('ICS 2025')*. ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

GPUs are popular platforms for accelerating data-intensive high-performance applications, such as graph analytics [2, 7, 24, 62], recommender systems [14], machine learning [23] and natural language processing [25, 56]. GPUs use massively parallel, high-throughput architectures that can provide high computing performance and memory bandwidth [1, 9]. GPUs are widely employed to accelerate data-intensive applications, which frequently operate on large-scale datasets, typically ranging in size from several gigabytes to tens of terabytes, and are likely to continue to increase in size in the foreseeable future.

The memory demands of modern data-intensive applications continue to outpace available GPU memory, despite increases in GPU memory capacity (e.g., up to 80 GBs for NVIDIA A100 GPUs [40] and 94GB for NVIDIA H100 GPUs [41]). In such cases, the application's memory does not fit on the GPU and typically resides on the CPU and programmers are responsible for moving data back and forth to the GPU (using *cudaMemCpy* or similar APIs), to ensure that the required data are available at the GPU when needed. This approach substantially complicates programming and requires careful optimization of data transfer operations, which is often not possible when applications are irregular with difficult-to-predict memory access patterns.

To ease these programming burdens, NVIDIA introduced the Unified Virtual Memory (UVM) [39] abstraction, where memory migration is automatically managed during run time. While UVM improves programmability and portability, it leads to substantial overheads [4, 5, 61]. Specifically, the performance of UVM and prior related works [19, 26, 64] is limited for generating as many requests as to utilize the available PCIe bandwidth fully. The latencies associated with handling transfer requests, as well as the inefficient use of available bandwidth which is an order of magnitude or lower than the GPU memory bandwidth [40] can substantially limit performance when the working memory does not fit within the GPU memory. We discuss the shortcomings of current approaches for extending the effective size of the GPU memory in more detail in Section 2.

As a result, the programmer is left with an unsatisfactory choice of (1) sacrificing programmability and manually managing and optimizing memory transfer operations between the CPU and GPU. This approach is cumbersome, hardware-specific, and may not be practical for applications where the memory access patterns are irregular; or (2) sacrificing performance and using UVM to ease programmability and portability by having the system manage the data transfers. In this paper, we introduce DREAM, a new GPU memory management system that supports both programmability and performance. It allows GPUs to directly manage memory, using Remote Direct Memory Access (RDMA) [6] to substantially reduce latency and improve data transfer throughput. This represents a departure in the philosophy of treating accelerators (including the GPU) as an offload device whose resources are managed by the CPU, to empower it to directly manage its application memory. DREAM eliminates CPU involvement and the latencies associated with CPU-mediated page fault handling. It also enables parallel fault handling to allow operations to overlap with ongoing GPU computations, thus improving overall system efficiency. DREAM incorporates several optimizations, including coalescing related data transfers and overlapping them with computation. In addition, DREAM employs highly efficient memory management and eviction schemes that deliver even higher advantages with increasing pressure on the GPU memory.

DREAM addresses several key challenges stemming from its programming model and the inherent limitations of current hardware, which lacks native support for GPU-driven memory management. On the GPU side, we have to build a runtime library that handles page requests/faults, manages the available memory space, and initiates communication to transfer memory pages. Specifically, DREAM implements a high-throughput, highly parallel, low-latency memory system to support on-demand page migration. The paged-memory system is optimized to coalesce the access requests and reuse the fetched pages. To handle oversubscribed cases where the application memory exceeds the available GPU memory, DREAM implements eviction logic to create room for newly fetched pages. We note that current CPU chipsets do not support GPU-managed data transfers between CPU and GPU memories without the involvement of host OS. To overcome this challenge, DREAM uses a network interface card to facilitate establishing RDMA connections which are used to mediate the transfer of the data from the CPU to the GPU. We provide more information about the design and implementation of DREAM in Section 3.1. We believe that future hardware support to enable direct RDMA between the GPU and the CPU can further improve the performance of DREAM.

We implement the DREAM software stack and evaluate it using Cloudlab [16]. We demonstrate DREAM on a number of benchmarks, demonstrating how it can directly support a range of applications without substantial programming overhead. DREAM achieves substantially higher performance than UVM on a number of GPU benchmarks with multiple datasets. It also outperforms optimized graph frameworks such as Subway [50]. The source code of DREAM can be found at https://github.com/nnurlan008/dream.

In summary, the contributions are as follows.

- We present DREAM, a new model for managing GPU virtual memory, supporting on-demand page migration without CPU intervention. DREAM leverages GPUDirect RDMA to enable GPUs to manage memory directly, substantially reducing page fault latency and improving PCIe utilization.
- We develop a high-level programming abstraction and software APIs to facilitate the integration of DREAM into existing applications.
- To support larger applications, we build a reuse-oriented paged memory allowing efficient eviction of pages when there is memory pressure on GPUs.
- We evaluate DREAM on a number of benchmarks, showing that it substantially outperforms existing solutions, including NVIDIA's UVM. DREAM achieves up to 4× speedup, which increases with the degree of memory pressure on the GPU.

2 BACKGROUND

In this section, we provide background on currently used paged memory systems for GPUs, which automate memory management between GPUs and the host CPU. We then explore how employing RDMA to manage virtual memory can alleviate the GPU memory wall problem and hide the memory access latency by allowing computations to coalesce with memory operations.

2.1 Unified Virtual Memory (UVM)

Unified virtual memory (UVM) creates a common virtual address space shared among all available memories in the system. This shared address space facilitates seamless data access and movement between devices and is utilized when memory is allocated using *cudaMallocManaged* [39]. For a system with a single GPU and CPU (host) shown in Figure 1, the allocation results in two separate page tables: one in GPU

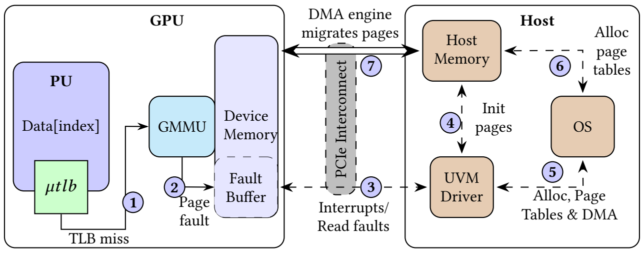


Figure 1: UVM architecture. PU refers to GPU processing units (Streaming Multiprocessors, or SMs on NVIDIA GPUs). SM-group level TLBs, and GPU-wide shared TLBs shown together as *μtlb* for simplicity.

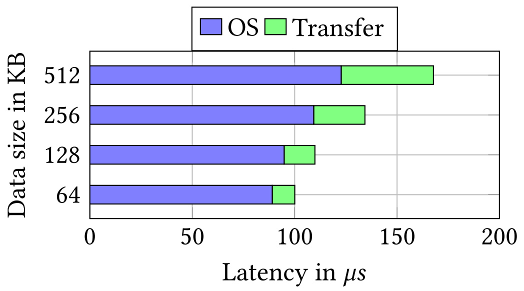


Figure 2: Breakdown of UVM page transfer latency. Note that host involvement overheads during the page fault are around 7× higher than the transfer time at 64KB page size.

memory and the other in host/CPU memory [3]. The UVM driver manages these page tables and handles page faults originating from the CPU side or the GPU side through the PCIe bus. The simplified page fault workflow from the GPU side is shown in Figure 1. When a thread accesses a remote page (a page residing in host memory), the *μ*tlb's [45], which are hardware units to cache recent page translations through multi-level TLBs, are checked for the address translations. When there is a TLB miss, the GMMU (GPU Memory Management Unit) is notified (1), which in turn writes the fault information into a fault buffer (2). GPU hardware threads treat this access request as any other outstanding memory access and, in general, experience a fault-and-stall condition [38], leading to disruptive interruptions in GPU computation leading to large inefficiencies that cannot be hidden using other warps.

To simplify conceptual presentation, only replayable faults are considered in this section. Each request triggers a hardware interrupt to the UVM driver (3) through a PCIe transaction. As the UVM driver retrieves a batch of faults from the fault buffer (3), it caches them in the host memory and initializes pages in the host memory (4). The UVM driver then informs the OS (5) to handle page table updates and TLB shootdowns (6). The host OS then directs the DMA engine to migrate the pages to GPU memory (7). Previous works have shown that the design of the UVM driver introduces delays in GPU application performance [3–5, 36, 65]. These delays are exacerbated by the lack of parallelism in CPU/OS in handling many requests coming from massively parallel GPU threads. We also analyzed the overheads of host involvement in page fault handling to be up to around 7× that of page transfer time, as shown in Figure 2, even at fairly large transfer sizes. Allen et al. [3] break down these overheads (fetching the faults from GPU fault buffer and preprocessing which involves de-duplication of the faults in the batch; sorting the faults by VA space, fault address, etc.), and show that they constitute around 12% of the overall cost. The biggest portion of the cost is spent on fault servicing which includes page table updates (both CPU and GPU), page eviction (under oversubscription pages have to be removed to make room for new pages), and other necessary actions. DREAM aims to reduce this overall cost by offloading memory management tasks to GPU and RNIC, allowing direct access that removes the CPU from the critical path.

2.2 Heterogeneous Memory Management (HMM)

HMM [30, 42, 51] is a Linux kernel feature designed to simplify memory management between different processing units, such as CPUs and accelerators. HMM provides programmer-agnostic memory management and on-demand memory access by allowing transparent page migration. Unlike UVM, HMM needs no vendor-specific driver calls for memory allocation. A memory region that has been allocated with *malloc* can be accessed by the accelerators on systems with HMM support. Similarly, HMM involves the OS and device driver for page fault handling. Both UVM and HMM are based on 4KB pages for *x*86_64 systems. Conversely, HMM does not natively support speculative prefetching, an optimization allowed by UVM that asynchronously migrates large GPU page (multiples of 64KB), to hide some of the page fault service cost. The responsibility of prefetching and asynchronous page migration still lies with the device drivers or userspace applications. Therefore, HMM is not an optimized method.

2.3 RDMA support and alternatives

Accelerator devices, such as GPUs, currently cannot initiate data transfers from host memory to device memory due to their design nature. UVM depends on the host OS to manage the on-demand memory accesses through the UVM driver. To enable device-initiated memory access and data transfers to/from device memory, our approach, on the other hand,

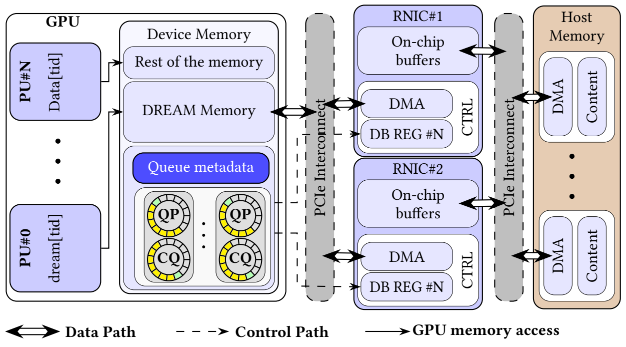


Figure 3: Schematic representation of DREAM design.

leverages RDMA-capable network interface cards (RNICs and NICs used interchangeably). This solution bypasses the need for CPU involvement, allowing the GPU to directly interact with the NIC for efficient data movement between host memory and device memory.

RNICs have limited capability processors [68] and enable direct access to host memory and other memories on remote machines across the network [32]. RDMA can access both local to a single system and remote memory over the network (InfiniBand, Ethernet/RoCE, etc.) [20]. Using RDMA read and write requests, a one-sided RDMA connection can be set up in which RNICs can read from and write to application memory without OS/kernel involvement. With GPUDirect RDMA [37], RNICs can also access GPU memory, enabling RNICs to mediate the transfer between a CPU and GPU.

3 DREAM SYSTEM DESIGN AND ARCHITECTURE

DREAM provides efficient device-managed memory access that automates memory management across GPU and host memory while achieving high performance. Programmers can use a high-level programming abstraction to gain access to DREAM, which aims to extend the available memory of GPUs or other accelerators to the host memory or available memory over the network. DREAM uses an RNIC as a mediator of requests between GPU threads and memory, as shown in Figure 3, since current systems do not support direct device-initiated page migration to/from host memory on demand[1] without the initiation of migration by the host OS. In this context, the available extended memory can either be the CPU/host memory, remote memory of other nodes in the cloud, or memory of other GPUs or accelerators connected through an RDMA-capable network (in this paper, we only explore the first alternative).

[1]GPUs do support direct access to host memory. However, this allows cache-line data transfer from host memory to GPU cache only and requires the pinning of the data on host memory.

DREAM needs to address a number of challenges: (1) **Supporting GPU-initiated memory transfers**: Systems are designed for the CPU to manage network-connected devices. To use the NIC for memory request handling for DREAM, GPU threads need to have access to the control resources of the NICs. Additionally, we need to make sure DREAM can efficiently generate parallel requests to be handled concurrently and efficiently by the RNIC. (2) **Supporting oversubscribed accelerator memory**: One of DREAM's goals is to provide efficient GPU memory oversubscription such that it can support workloads with larger memory requirements than are physically available in GPU memory. Since memory is managed by the GPU, it must be able to evict pages to make room for newly fetched pages. DREAM must ideally map the pages in host memory to GPU memory, which can possibly prevent early eviction of pages from GPU memory before they are used under memory oversubscription. (3) **Programming Abstraction**: Since the GPU threads are not typically designed to make memory accesses/requests by themselves, DREAM needs to offer high-level abstractions that conceal its complexity and simplify the integration of DREAM into existing GPU applications for programmers. In the remainder of this section, we first present an overview of DREAM, and then discuss how we address these three challenges.

3.1 System Overview

In this section, we explain the high-level design of DREAM. In systems such as UVM that rely on OS page fault handling [8, 18, 19, 26, 28, 34, 67], the OS is primarily responsible for memory allocation, controlling virtual-to-physical address translations, and page table updates. DREAM aims to shift the memory management to the GPU, removing the host OS from the critical path, and has to reimplement the page management functionality on the GPU side. Since current hardware does not allow the GPU to initiate RDMA transactions from CPU memory, DREAM uses an RNIC to facilitate the transfer. RNICs support one-sided RDMA connections to the CPU, which allow a device to move pages from other memories directly (without OS involvement).

The incorporation of RNICs extends GPU accessibility beyond local host memory, facilitating direct access to both remote host memory and remote GPU memory across the network. In this study, we focus on a specific scenario where DREAM is leveraged to enable GPUs to efficiently access local host memory on demand, thereby minimizing latency and optimizing data locality in heterogeneous computing environments.

As shown in Figure 3, DREAM enables GPU threads to directly communicate with the RNIC by submitting work/page requests to the queue pair (QP). GPU threads are notified

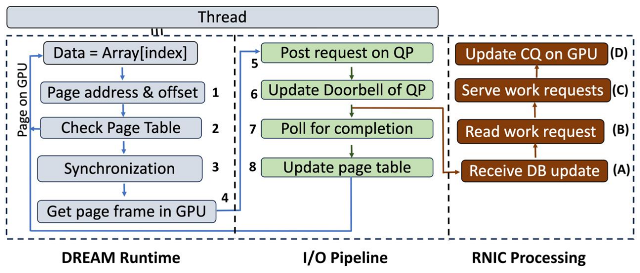


Figure 4: DREAM system workflow for a single thread

Listing 1: Vector addition with DREAM

```
__global__
void vectorAdd(dream_ptr<float> *A, dream_ptr<float> *B, dream_ptr<
      float> *C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
       C[i] = A[i] + B[i];
    }
}

int main(){
    dream_ptr<float> host_a, host_b, host_c;
    dream_ptr<float> *dev_a, *dev_b, *dev_c;
    // allocate data
    host_a.allocate(size on CPU, size on GPU);
    ...

    // populate the arrays
    for (size_t i = 0; i<size on CPU; i++)
       host_a[i] = rand_float_generator();
    ...

    // allocate dev_a, dev_b, dev_c on GPU memory
    cudaMalloc((void **) &dev_a, sizeof(dream_ptr<float>));
    ...

    // copy metadata from host_a, host_b, host_c to dev_a, dev_b,
         dev_c respectively to make them accessible from GPU
    cudaMemcpy(dev_a, &host_a, sizeof(dream_ptr<float>),
         cudaMemcpyHostToDevice);
    ...

    //launch kernel - size is actual size of the buffer; size on CPU
    vectorAdd<<<#ofblocks, block_size>>>(dev_a, dev_b, dev_c, size);

    // bring data back to CPU if needed
    host_c.toCPU();

    // destroy a, b, and c:
    host_a.destroy();
    ...

    return 0;
}
```

of the request completion through completion queue (CQ) entries inserted by the RNIC once a request has been serviced. DREAM uses a memory region called DREAM memory within device memory with device page tables and page maps efficiently managed by parallel GPU threads. The GPU hosts the DREAM memory region, queue metadata, and parallel RDMA queues.

An important part of the abstraction is the *DREAM::class* structure, which enables direct integration of DREAM into existing GPU kernels. We show an example in Listing 1 for a vector addition kernel; the code is minimally modified to include DREAM's array-like data structure. The DREAM programming model provides users with the flexibility to modify data on the CPU and launch kernels with DREAM arrays - *dream_ptr<float>*. In the current implementation, the responsibility of determining the GPU memory allocation is delegated to the user. By default, the allocated GPU memory is set to either match the size of the data on the CPU, the actual size of the allocated data, or the maximum available GPU memory in cases where the required memory exceeds the available capacity. Once kernel execution begins on the GPU/accelerator, data is migrated to device memory on demand. After the kernel execution finishes, users have the option to either transfer the entire dataset/buffer into CPU memory or access data on demand from the CPU. Upon completion of the computation, the DREAM objects are properly deallocated, and resources are released.

Figure 4 shows how a thread can access the page in DREAM. First, the thread that accesses the DREAM::class buffer at a specific index calculates the page address/number and the offset (1) and checks the page table for this page address (2). If the page is currently in the GPU memory, the thread can safely access the offset within the page. Otherwise, DREAM runtime synchronizes all threads that access the same page (3). Within the synchronization block, a leader thread is selected to handle the page fault, which starts with obtaining a page frame in GPU memory (4). Once a page frame is obtained, the leader thread prepares a work request for the page, which includes a unique request number, page frame address, host memory address, remote key for host memory, and the QP ID. Once the leader thread inserts the request into the QP (5) and updates the doorbell register (6), it starts polling the CQ entry associated with the request number (7). More details are in Section 3.3.

On the RNIC side, when the doorbell update is received, the RNIC starts fetching the work requests from the QP in the GPU memory. The work requests for the page requests from the GPU threads are served by the RNIC. For each page request, the RNIC fetches the page from the CPU memory and sends it to the GPU memory (DREAM memory in Figure 3). Once the page is delivered to the assigned page frame, the RNIC updates the related CQ entry to notify the leader thread that the work request has been serviced.

Note that the RNIC processing component can be substituted with alternative interfaces or hardware by appropriately adapting the I/O pipeline. For instance, incorporating FPGAs could enhance local memory access performance, while leveraging SmartNICs may provide additional benefits

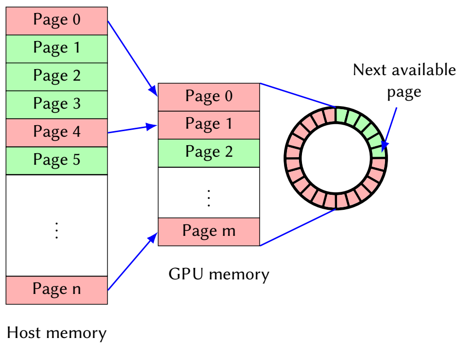


Figure 5: DREAM page mapping: Host memory contains all pages. GPU memory is organized as a circular page buffer. Red represents mapped pages, and green unmapped pages.

in scenarios involving remote memory. Such hardware substitutions have the potential to further improve the efficacy and versatility of DREAM as a comprehensive solution.

3.2 Efficient RDMA I/O Management

To achieve high transfer throughput, DREAM uses a memory-efficient, streamlined I/O pipeline shown in Figure 4. This pipeline enables GPU threads to manage page requests; post requests and poll for their completion. The DREAM runtime also has the GPU leader threads directly accessing RNIC I/O resources and updating page table entries.

Sending a page request can be divided into two actions: inserting a work request for the page and ringing the doorbell which means writing the number of the request and index of the queue to the doorbell register. As the doorbell registers are located in the system memory, updating them for a single request would cause a serialization, thus, increasing the latency if the number of queues is not sufficient.

The leader thread gets assigned a queue index that identifies which QP and CQ the leader will use for posting and polling, based on SM number. The leader atomically gets a number for the request called *post_number* that it uses as an ID of the page fault. Faults are handled in batches. If the post number is less than the fault batch size, then the leader can continue to insert the request into the send queue with *post_number*. Otherwise, it must wait for the current batch to finish. For a fault batch, several leader threads insert requests for different pages. However, only one randomly selected leader (the one that atomically gets the lock of the queue) updates the doorbell. Once the queue is locked no other request can be inserted.

All the leader threads within the batch atomically increment a variable called *batch_counter* to coordinate and ensure the completion of their insertions into the send queue, which serves as the sole communication channel for one-sided data transfers between the GPU and host memory. . If this variable matches with the max post number, then all the leaders have finished the insertion of page requests into the send queue. At this stage, the leader with the queue lock can update the doorbell. After the update of the doorbell, all the leader threads in the batch start polling on completion queue entries corresponding to *post_number*.

The period during which leader threads wait for completion is consistent, depending on the NIC, and threads release the lock once their requests are fulfilled, leaving no opportunity for security channels or performance slowdown attacks.

The queue depth to achieve the max available throughput is covered by Little's law [33], $L = \lambda * W$, where L, λ, and W represent the average queue depth, access latency, and target throughput, respectively. The target throughput is $12GBps$ as we are using a PCIe-3-based machine. The latency λ is $23\mu s$ empirically estimated on the testbed we use in our experiments. From Little's law, we obtain an average queue depth of 36 for an 8KB page and 72 for a 4KB page. Therefore, optimal performance can be achieved with a batch of one fault with more than $72(23\mu * 12GBps/4KB)$ parallel queues (for 4KB pages and, similarly, 36 queues for 8KB pages).

3.3 Device Virtual Memory Management

Systems that utilize OS virtual memory mechanisms for data migration implement several key steps to facilitate on-demand page migration. These steps include page table updates, address translation, and other related operations, all of which occur during application runtime, as illustrated in Figure 1. However, these processes often involve critical sections with significant serialization, resulting in increased fault service latency. In contrast, DREAM leverages RNIC hardware for address translation, enabling a substantially higher degree of parallelism and in-flight request processing by bypassing the host OS. The device memory page table is updated directly by the device upon the completion of a work request on the RNIC. Implementation details are provided in Section 4.

Within the DREAM runtime, the host virtual memory can be considered as "Physical Address Space" and GPU virtual memory as "Virtual Address Space" as shown in Figure 5. The host memory is assumed to be the backup and contains all the application data. GPU memory can host all the data and sometimes becomes limited as the workload sizes get bigger than GPU memory.

Page mapping. The main goal of DREAM runtime is to dynamically and efficiently map the pages from 'physical' to 'virtual' address space which can be defined as mapping the virtual page on host memory to a virtual page on GPU memory as shown in Figure 5. Upon a request from GPU threads, RNIC hardware has to bring the page to GPU memory (DREAM memory in Figure 3). After all the threads

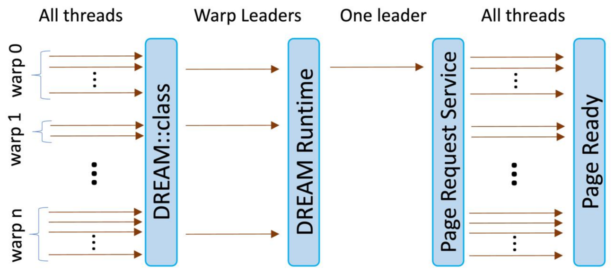


Figure 6: DREAM runtime workflow. Threads access DREAM::class and warp leaders are forwarded to DREAM runtime. Obtaining a page involves acquiring a mapping and posting and polling for a page request.

complete the read/write operation on the page, it can be safely evicted if necessary. The page mapping is stored in device memory for fast access and dynamically modified by GPU threads during application execution. For current implementation, DREAM keeps page mapping and tables locally on device memory. However, for future implementations, this can be changed or even maintained by a new interface/hardware if coherence is required across multiple accelerator devices.

Eviction scheme. The DREAM implementation specifically aims to improve the ease of programming for developers and eliminate the need for partitioning of the workload and the manual transfers of the partitions that are needed when the workload size is bigger than the available GPU memory. To achieve stable performance as the oversubscription level increases, the efficient FIFO page eviction mechanism is implemented.

The GPU virtual memory can be viewed as a circular queue (ring buffer) with a global head cursor depicted in Figure 5. Each entry in this queue represents a page mapping. When the leader thread handles the page fault, a new page mapping that maps the host page to the GPU page is established. The leader thread is responsible for establishing the mapping through DREAM runtime library. It is important to note that the leader thread acquires the mapping atomically and is restricted from arbitrarily mapping to any GPU page frame. After mapping is established, the leader thread checks if the page frame is mapped to another page. If the page frame is already mapped to another page, it waits for that page to be released. This happens through checking **reference counter** which shows the number of warps that currently access/need the page. Once the reference counter becomes zero, the leader thread immediately evicts the page to the host with a write-back request if necessary and creates the new mapping.

Page access synchronization. We use the opportunistic warp-level synchronization primitive [11], `__match_any_sync`, to select a leader within warp threads with the same `__activemask`. DREAM runtime also incorporates inter-warp coalescing, considering that there might be more than one warp that accesses the same page; within all the leaders from those warps, only one leader will lock the page entry, as illustrated in Figure 6. Page fault or page request in Figure 6 involves getting a mapping, triggering a work request on RNIC, and eviction of another page if necessary. Once the page request is complete, all threads can continue processing the data.

3.4 Comparisons with UVM and Bulk Transfer

Unified virtual memory makes the host and GPU memories virtually connected, allowing the device code to directly access the system memory and provide a programming-agnostic development experience. The bulk transfer involves partitioning the data and transferring the partitions to GPU memory by CPU. There are **several clear advantages of DREAM over UVM and bulk transfer approaches**.

Firstly, UVM transfer size is 4KB (for x86_64), and 60KB is asynchronously transferred due to speculative prefetching to complete the size to GPU page size and boost the performance [3, 52]. The eviction size is 2MB [3], which is called a virtual address block (VABlock). This can sometimes be the bottleneck for oversubscription, as the newly fetched page can be evicted within VABlock. However, since in DREAM, GPU threads have direct access to the page table and can monitor the page status and reference counter, eviction becomes much more efficient. For example, if the page is write-intensive, DREAM can delay its eviction and evict one of the least needed read-intensive pages.

Secondly, to optimize the UVM, the application developer should statically specify the access hints before the application starts running on the GPU. Thus, UVM cannot leverage dynamic memory optimizations, making it more developer-dependent. On the other hand, DREAM can benefit from dynamic and efficient page mapping optimizations, such as reference counters for each page.

Furthermore, a widely adopted approach involves partitioning the dataset and transferring the resulting partitions into GPU memory for computation. This method leverages the high memory bandwidth of GPUs to enhance application performance. However, it necessitates that developers manually partition or tile both the data and the algorithm, which can pose significant challenges. In particular, for data-dependent applications such as graph processing, recommender systems, and data analytics, identifying an optimal partitioning strategy is often complex and non-trivial. Consequently, DREAM offers a compelling alternative by enabling on-demand data access, thereby alleviating the burden of manual partitioning.

4 DREAM IMPLEMENTATION

Constructing Virtual Memory: As DREAM relies on virtual memory, a large CPU/host memory region is allocated using *malloc* to host application data and registered to RNIC *ibv_reg_mr* verb with appropriate access flags.The access information of host memory which includes remote keys, starting address, and the length of the allocated/allowed region, is copied into GPU memory for easy access from GPU threads during runtime.

Access to RNIC from GPU Threads: To achieve high throughput, GPU threads are expected to solve page fault requests fast. The main high-latency part is where the GPU threads start inserting the requests into the send queue and update the doorbell as these steps involve host memory accesses. Prior work(s) [58] proposed to map the RNIC resources such as QP and CQ buffers and doorbell registers to boost the performance of network-related applications on GPUs. However, GPU threads need to access these resources through PCIe bus, and due to high access latencies and lack of efficient concurrency, this approach cannot achieve high throughput with low granularities (page size in our case). Instead, we allocate completion and queue pair buffers on GPU memory using *cudaMalloc* and associate these buffers with the RNIC resources, namely, *ibv_cq* and *ibv_qp*. To this end, we modify the *rdma_qp_create* and *ibv_cq_create* functions in the newest Mellanox Infiniband driver to include user-defined buffers similar to the GPUrdma [12]. This makes sure the GPU has access to the RNIC's necessary buffers in its memory.

To give GPU access to the doorbell registers, we map them to CUDA address space using GPUDirect Async [43]. For this, we register the doorbells using *cudaHostRegister* API with *cudaHostRegisterIoMemory* flag to map them to GPU's BAR space and get device pointers using *cudaGetDevicePointer*, enabling GPU threads to ring them on demand.

Discussion: Multi-kernel support. In the current DREAM implementation, the GPU memory is statistically allocated. The mapping of the pages from host memory to GPU memory happens dynamically during application run time. This paper evaluates DREAM mostly on single-kernel applications and multi-kernel with kernels launched back-to-back. However, the current implementation can be easily extended for concurrently launched kernels in which a stream of concurrent kernels accesses the large dataset on demand.

Multi-GPU processing. DREAM currently supports 2 GPUs and 2 NICs enabling multi-GPU co-processing in the system. The GPUs can share the NICs for data transfer and concurrently work on the data independently without requiring the programmer to manually create and transfer partitions to the GPUs, separately to amplify the read throughput and access data on demand.

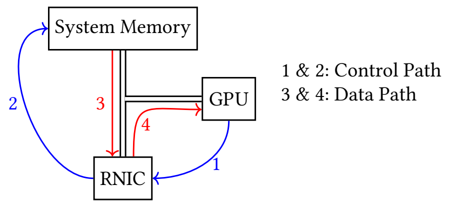


Figure 7: Configuration of r7525 node in Cloudlab [16]. As 3 & 4 share the same PCIe bridge connecting NIC to the system, the available bandwidth drops to 8GBps, halving PCIe 3 bandwidth.

4.1 Limitations

We evaluate DREAM on an r7525-type node in Cloudlab [16]. The system configuration is shown in Figure 7. The performance of DREAM is specific to this platform and may be sensitive to the specific hardware. The node consists of 2 GPUs and 2 NICs that are connected to the root complex through dedicated bridges. The page migration works as follows. While the kernel is executing on the GPU, any page request will send/trigger a work request to the NIC (1). Upon fetching the work request, NIC sends a request to memory (2). Upon the request, memory DMA sends the requested data to NIC (3). NIC finally sends the requested page to GPU memory once received the data (4). The first drawback in this transfer mechanism is the interruption of the NIC as it is located on the data path. Another downside is that the data incoming to and outgoing from the RNIC shares the same PCIe bridge channel, decreasing the one-directional bandwidth to half of the available bandwidth. To overcome this issue, one possible solution is to modify the requests from NIC to memory such that the page is delivered to GPU memory directly from system memory. However, with modern NICs, this approach is not feasible, as it necessitates modifications to the closed-source NIC firmware. Alternatively, we use both RNICs available on the node for data transfers, increasing the transfer throughput to the maximum available bandwidth as shown in Figure 8. DREAM also assumes that host memory is pinned and that there is sufficient host DRAM capacity to accommodate the working set.

This limitations can be relaxed in different ways, for example pinning only the hot pages, and relying on page faults for pages that are not pinned. We hope to explore such ideas in future work.

5 EVALUATION

In this section, we provide the performance evaluation of DREAM over different benchmarks and applications and compare the results with other state-of-the-art. The experiments have been conducted on r7225 nodes of Cloudlab [16], a testbed with the configuration shown in Table 1.

Table 1: System configuration for experiments

Component	Specification
CPU	2× AMD 7542 (32 cores, 2.40 GHz)
GPU	NVIDIA Tesla V100 32GB
RAM	512GB 3200MHz DDR4
NIC	NVIDIA Mellanox ConnectX-5 25Gbps & ConnectX-6 100Gbps
Software	Ubuntu 22.04 LTS, NVIDIA Driver 535.183.01, CUDA 12.2

5.1 Comparison with GPUDirect RDMA

First, we compare the performance of DREAM with NVIDIA GPUDirect RDMA [37] on a simple data transfer benchmark with different request sizes. Request size is given as input in the scatter-gather entry and defines the data length to be transferred from host/system memory to GPU memory through RNIC. The benchmark includes the transfer of 12GB of data from host memory to GPU memory with different request sizes ranging from 4KB to 1MB. In the case of GPUDirect RDMA, the transfers are initiated from 16 concurrent threads in the CPU. For DREAM, each GPU warp initiates a transfer with the same request size of GPUDirect RDMA corresponding to consecutive addresses. DREAM launches 16 warps in each of the 84 SMs simultaneously. Each warp is assigned a page. The benchmark is conducted using a single NIC and 2 NICs. As shown in Figure 8, DREAM can achieve the max usable bandwidth available for transfers through a single NIC, which is 6.5 GBps, even with a 4KB page size. Conversely, GDR can reach the maximum available bandwidth utilization after requests of 512KB. DREAM can keep a stable performance with almost all page sizes and fully utilize the PCIe 3 bandwidth with 2 NICs.

In the case of UVM, we optimize the memory access patterns such that each warp accesses 64KB of consequent data and uses memory hints (memadvise) for optimization. UVM does not allow flexible page sizes. Therefore, we can only report the average throughput, which is observed to be around 6 GBps, achieving only 50% of the available bandwidth.

5.2 Graph Workloads

In this section, we evaluate DREAM on different graph analytics workloads with various graphs listed in Table 2. All

|E|: number of edges, |V|: number of vertices

Dataset Name	Abbr	|E|	|V|	Size (GB)	
				Edges	Weights
GAP-Urand [29]	GU	4.29B	134.2M	16.0	16.0
GAP-Kron [29]	GK	4.23B	134.2M	15.7	15.7
Friendster [63]	FS	3.61B	65.6M	13.5	13.5
MOLIERE [59]	MO	6.67B	30.2M	24.8	24.8

Table 2: Description of graph datasets.

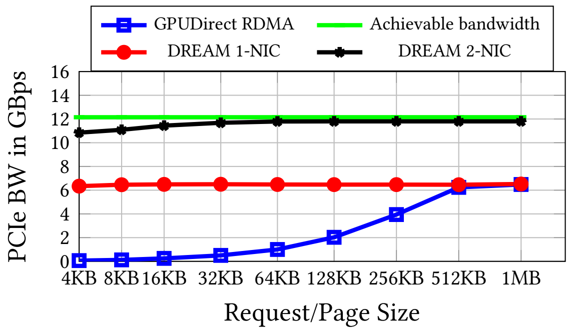


Figure 8: Achieved PCIe bandwidth. DREAM saturates PCIe bandwidth even at 4KB page size. GPUDirect RDMA can only saturate the interconnect after 512KB granularity.

evaluated graphs are from the SuiteSparse Matrix collection [13]. The main goal of the DREAM paged-memory system is to provide better results over the UVM solution and approaches using bulk transfer. To this end, we choose a well-optimized open-source UVM solution [35] as a UVM baseline (**U**). Additionally, we also compare results with Subway [50], which is an optimized graph framework that first partitions and preprocesses the graph to create subgraphs of smaller graphs on the CPU and then transfers the partitions to the GPU for traversal.

For graph workloads, we choose Breadth First Search (**BFS**), Connected Components (**CC**), and Single-Source-Shortest-Path (**SSSP**). For BFS and SSSP, the application is executed with more than 100 source vertices with at least 2 neighbors, and we take the average of the execution times. The reported time for each approach is the average application execution time and UVM memory advise API time if applied. As Section §3.1 describes, the integration of the existing state-of-the-art implementation into DREAM is easy. We use 1 NIC and 2 NICs with 84 queue-pairs of 64 entries.

For DREAM (**D**), we evaluate two versions –baseline and optimized– for graph workloads and compare them against UVM alternatives [35] and partitioning-preprocessing-based approaches [50]. The baseline approach (**1N**) uses CSR representation of graphs and uses only 1 NIC. We observed that the imbalance in the number of neighbor lists can become a bottleneck for DREAM. Therefore, for the optimized version (**2N**), we create a new representation called Balanced CSR and use 2 NICs to be able to exploit the GPU PCIe bandwidth. We use this representation for graphs with power law edge distribution. For instance, the FS graph exhibits a maximum vertex degree of approximately 5,200, whereas the GU graph has a maximum degree of only 68. However, GK and MO have a maximum vertex degree of around 7.5M and 2.1M neighbors, respectively. This means the number of

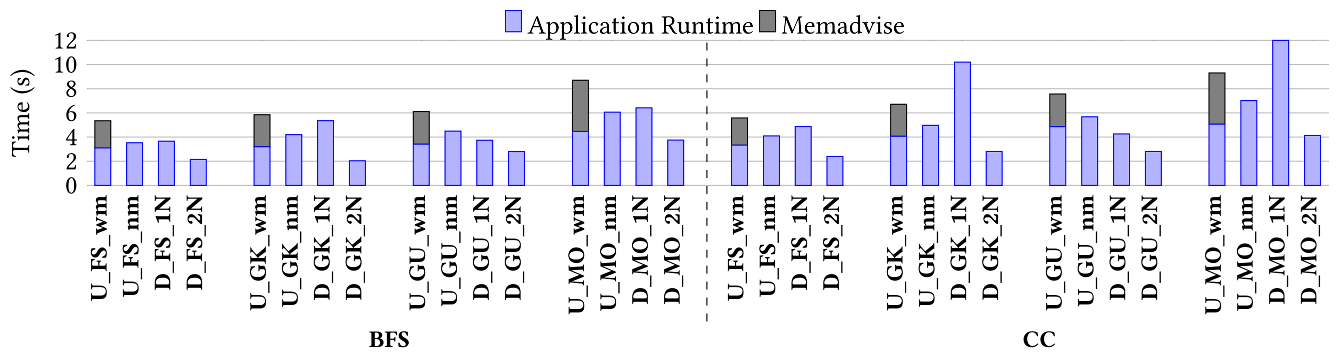


Figure 9: Graph workload evaluation results. 'U': UVM, 'wm': with memory advise hint (applies to only UVM), 'nm': no memory advise hint, 'D': DREAM, '1N': 1 NIC in CSR, '2N': 2 NIC in Balanced CSR. DREAM implementation uses 4KB-sized pages. DREAM's overall performance is 1.4× for BFS and 1.5× for CC better than UVM solution.

page requests per thread can be high as high degree nodes are visited, resulting in request serialization. To prevent this serialization, we introduce Balanced CSR representation, a modified version of CSR that stores edges in chunks of equal numbers as shown in Figure 10. This format partitions neighbor lists –particularly long ones– into smaller, equally sized segments and saves starting indices of these chunks in the offset list. Each index in the vertex list indicates the starting vertex. This format enables different threads or thread groups to access and process these segments in parallel. This parallelism improves load balancing and mitigates bottlenecks associated with skewed degree distributions as the number of serial page requests per thread or thread group is reduced. The memory overhead of Balanced CSR is up to 400 MB for the graphs in Table 2, while substantially improving the performance.

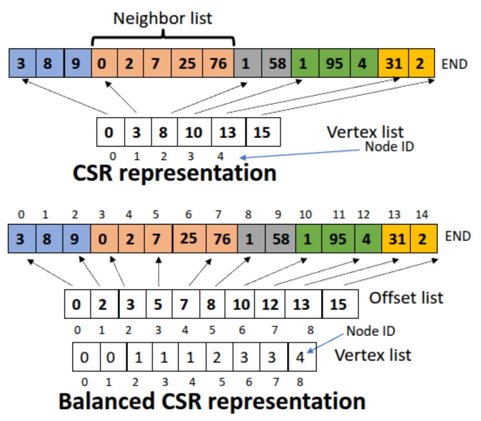


Figure 10: CSR vs. Balanced CSR. The Balanced CSR representation allows multiple threads to concurrently process the same neighbor list, thereby reducing traversal latency for DREAM.

Benchmark	Graph	Total Time (s)		Speedup
		Subway	DREAM	
BFS	GK	3.86s	2.04	1.89×
	GU	3.13s	2.80	1.12×
	FS	2.52	2.15	1.17×
CC	GK	4.73s	2.81	1.68×
	GU	5.21s	2.80	1.86×
	FS	3.24s	2.40	1.35×

Table 3: Performance comparison to Subway [50]

Baseline UVM [35] uses optimizations for coalesced memory accesses and assigning each warp some number of vertices (instead of a single thread single vertex model) to traverse. It is also optimized (**wm**) for read-only accesses by setting UVM memory as *cudaMemAdviseSetReadMostly* using *cudaMemAdvise()*. With this flag, the UVM driver will make read-only copies of the pages on GPU memory, substantially reducing the page fault service cost and page migration time. Results shown in Figure 9 demonstrate that the use of this flag increases the application performance by at least 25% over the one without memadvise (**nm**). Although the application runtime decreases significantly, setting this memory advice hint incurs substantial initial delay, which is also reported in our evaluation results but is not included in the speedup calculation. UVM performance does not benefit from the Balanced CSR representation, as it has speculative prefetching and page request batching.

Evaluation. Figure 9 shows that DREAM's performance improves by an average of 1.5× when using the second NIC, achieving an average 1.4× improvement for BFS and 1.5× for CC. As DREAM provides a transparent coding experience for application developers by abstracting away data transfers, it is essential that its performance matches or exceeds that of solutions utilizing explicit data transfers, such as *cudaMemcpy*. In this sense, we compare the results of the BFS and CC applications with Subway [50]. Subway provides a solution for large graph traversal by first pinning and processing

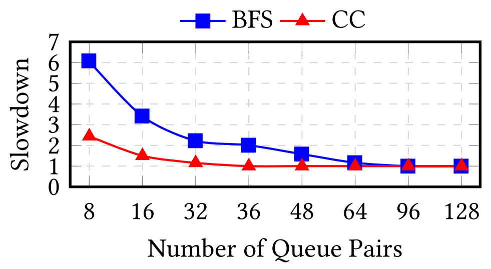


Figure 11: Sensitivity to number of QPs and CQs

the graph on the CPU to create smaller subgraphs of active nodes that can fit into GPU memory and transferring these subgraphs to GPU for traversal. As Table 3 shows DREAM can achieve an average speedup of 1.4× for BFS and 1.6× for CC over Subway [50]. Subway is limited to graphs of less than 2^{32} vertices; it cannot support the **MO** graph.

Impact of queue count. To understand the effect of the number of queues on the performance, we measure the slowdown in the performance of CC and BFS as the number of queues is changed. As shown in Figure 11, the applications start demonstrating near-optimal performance as the queue count exceeds 48.

SSSP with limited GPU memory. To understand the effect of GPU memory limitation on application performance, we evaluate DREAM and UVM for SSSP with GPU memory limited to 16GB. Both approaches need to evict data from the GPU memory to bring a new page. As Figure 12 presents DREAM achieves an overall speedup of 1.9× on SSSP performance on limited GPU memory. As the eviction size in UVM is coarse-grain (2MB), it can evict the data that has not been accessed by GPU threads and might be needed later. However, since DREAM efficiently leverages the reference counter for each page and has an eviction size of 4KB, DREAM reduces the redundant data transfer by 1.8×.

5.3 Transfer-bound Applications

In this section, we evaluate performance on CUDA benchmarks: MVT (matrix-vector transpose), ATAX (matrix transpose and vector multiplication), BIGC (big compute), and VA (vector add). The performance of these applications depends on efficient data transfer as they involve simple computations. The workloads in this benchmark suite can fit into GPU memory. MVT, ATAX, and BIGC benchmarks [21] involve the transpose of a matrix that requires memory accesses through the columns of the matrix reducing the spatial locality. For these applications, DREAM performs up to 4× better than UVM when using two NICs and up to 2× with one NIC, achieving much better PCIe utilization as shown in Figure 13. Vectoradd given in Listing 1 adds two DREAM buffers with two billion floating-point elements and stores

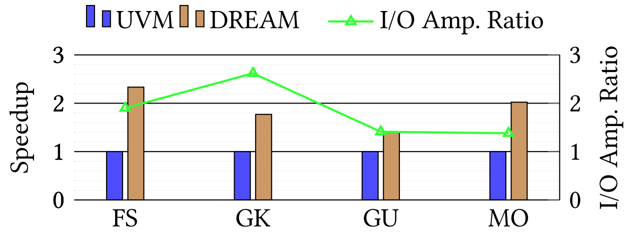


Figure 12: SSSP with 16GB GPU memory.

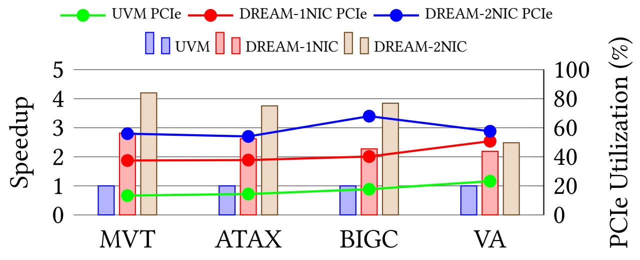


Figure 13: Performance (bars) and PCIe Utilization (lines) for UVM and DREAM

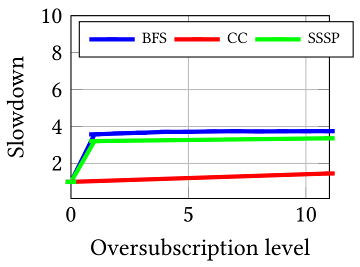
the results in another buffer. DREAM can provide just over 2× speedup and better PCIe utilization as depicted in Figure 13. When we restrict the GPU memory for VA, we obtain around 1.7× speedup over UVM. The lower speed-up result is achieved because DREAM is not optimized for efficient writebacks under GPU memory pressure; specifically, we have not yet implemented asynchronous writebacks, resulting in increased latencies on these operations.

5.4 Oversubscription Analysis

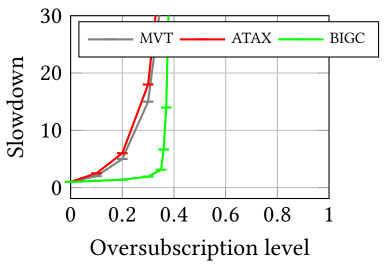
In this analysis, the goal is to observe the performance slowdown of different applications when the GPU memory gets smaller than the workload size. For this, we artificially limit the GPU memory and keep the workload size fixed. The pressure on the GPU memory compared to the workload size can be defined as in (1)

$$\frac{\text{Workload Size}}{\text{Available GPU memory}} - 1 \tag{1}$$

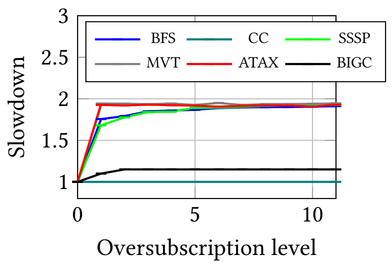
One of the goals of DREAM is to alleviate the programmers from having to handle manual data partitioning and transfer by offering efficient oversubscription of GPU memory with stable performance. The existing approach is UVM oversubscription. Prior works [3, 27, 55] have shown the degradation of application performance with UVM oversubscription. The performance degradation can be directly related to the inefficient eviction mechanism which has a large page size of 2MB and can often evict the pages even before they are accessed by the GPU threads under memory pressure.



(a) UVM for graph analytics



(b) UVM for matrix-vector operations



(c) DREAM approach

Figure 14: Effect of oversubscription on GPU memory for different algorithms with UVM and DREAM.

DREAM, on the other hand, implements a FIFO-based reference priority eviction mechanism as described in Section §3.3. This eviction scheme happens when there is not enough memory for DREAM runtime, and the thread needs to get a new page frame that maps the page in host memory to a page in GPU memory. In this case, the runtime atomically gets a new page frame and checks the reference counter. Once the counter becomes zero, no thread needs to access the page; it is safe to evict the page, and the page frame is assigned to the newly fetched page.

We run several different applications under increasing oversubscription levels. The key observation from Figure 14 is that DREAM can provide consistent and predictable performance even under highly pressured GPU memory. However, with UVM, the performance is mostly dependent on the data access patterns in the application. UVM can slow down the performance of graph applications by 4× compared to around 2× slowdown of DREAM. With MVT, ATAX, and BIGC, Unified Virtual Memory (UVM) incurs exponential slowdowns due to column-wise access patterns, which lack spatial locality. As a result, these applications fail to benefit from UVM's speculative prefetching mechanisms, leading to significant performance degradation. DREAM, on the other hand, can keep the performance stable by introducing up to 2× slowdown. The similar jumps in the slowdown for all applications in DREAM happen because the leader threads call eviction functions that introduce additional latency to page request time.

5.5 Query Evaluation Benchmarks

For query evaluation, we compare the performance of RAPIDS [46] by NVIDIA and custom query search with UVM and DREAM. RAPIDS is a framework to make query searches over the datasets using GPU. We provide 5 queries for the Chicago Taxi Trips Dataset [10]. The main question in the query is to find *"The average dollar per mile a driver makes in trips that take longer than 9000 seconds."* To find the ultimate result, firstly, (Q1) we find the total miles in such trips (>9000 seconds); secondly, we find the total fares (Q2), then total extras (negatives) (Q3), then the total tips (Q4), and finally, total tolls (Q5). For all comparisons, the dataset is loaded into the system memory. To optimize the run time of RAPIDS, pinned buffers are employed. DREAM and UVM rely on host memory as the backup and make on-demand page requests to the data in host memory.

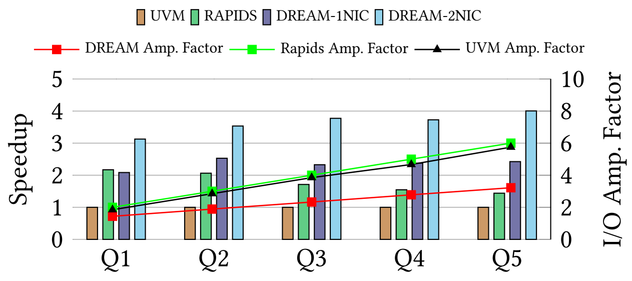


Figure 15: Query evaluation performance. DREAM uses 4KB pages. UVM uses a minimum size of 64KB (including base page - 4KB + prefetch). (0.08% sparsity)

Results. Figure 15 shows that UVM performs on average 1.5× and 3× slower than RAPIDS and DREAM, respectively, with no advantage for I/O amplification. DREAM benefits from high-throughput, smaller granularity page transfers leading to a performance improvement of over 1.5× with one NIC and 2.5× with two NICs over RAPIDS (Q5). Although RAPIDS can benefit from high-bandwidth data transfers due to pinned buffers, it needs to transfer the whole data (entire columns) for processing as it lacks on-demand access, leading to significant I/O amplification (redundant data transfer). UVM can have high I/O amplification as the page size is large (4KB page fault + 60KB speculative prefetching). However, DREAM halves I/O amplification for a sparsity of 0.08%.

5.6 Discussion

Overheads. DREAM allows GPU threads to interface directly with the NIC. Specifically, both page mappings and

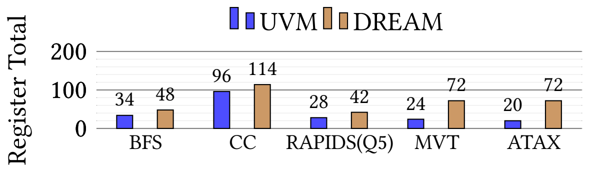


Figure 16: Register use per thread for UVM and DREAM. No register spilling occurs.

page tables are stored in GPU memory, enabling faster access for GPU threads. In total, DREAM incurs memory overhead of up to 8MB for page tables and mappings and 2MB for NIC control resources. In addition, as Figure 16 illustrates, using DREAM on GPUs does not cause any register spilling for the applications studied in this paper.

Additional Insights. DREAM is a practical solution for accelerators to access extended memory efficiently. Its performance could be further improved with hardware support like direct memory transfers between accelerators and the CPU. We addressed this via the RNIC, which also enables remote memory access and supports memory disaggregation across GPU clusters.

6 RELATED WORK

Unified memory and storage. Prior works [8, 18, 19, 26, 28, 31, 34, 47, 64, 66, 67] have been proposed to enable efficient memory access through unified memory and storage with UVM. For example, Choi et al. [8] proposes a new system for efficient unified memory systems for multi-GPU systems. Koukos et al. [28] propose a novel scheme to facilitate heterogeneous systems (CPU-GPU) with unified virtual memory. Ziabari et al. [67] proposes a new hardware-based unified memory hierarchy for multi-GPU systems. Zhang et al. [66] presents a unified GPU memory and storage architecture for tensor migration in deep learning workloads. Markthub et al. [34] extends the UVM to be able to make page-fault requests to storage devices. Other works [19, 26, 64] mainly leverage software modifications to improve the UVM performance. The common part among these works is that they rely on the host OS page fault handler for data migration. Therefore, performance is limited due to the lack of page fault handling parallelization and OS involvement delay. In addition, the recently introduced Grace-Hopper Superchip [17, 52] supports unified memory with hardware page tables and creates coherent CPU-GPU memory by bringing together the Grace CPU and Hopper GPU through NVLink-C2C [44]. However, it relies on a new and different CPU design that supports NVLink connection becoming an expensive hardware-dependant solution.

Enabling direct storage access for larger memory. Some prior works [48, 53, 57, 60] enable direct storage access for GPUs. Silberstein et al. [57] proposed POSIX-like file system APIs for GPU programs by integrating CPU's cache buffer into GPU memory. Active pointers by Shahar et al. [53] are abstractions similar to memory-map to enable GPU threads to access storage devices. Qureshi et al. [48] recently proposed a software cache in GPU memory to enable on-demand, fine-grain, and high-throughput access to storage. DREAM differs from these works as it delivers efficient on-demand access to system memory, rather than storage and provides efficient oversubscription of GPU memories.

DREAM can be thought of as a form of memory disaggregation, but driven by GPUs. The use of RNIC empowers accelerators/GPUs to efficiently access the remote memory on demand. Previous works [15, 49, 54] aim to provide efficient fine-grain manipulation of remote data for CPU workloads. DREAM shares similarities with these works in that they also employ similar user-space mechanisms for managing memory accesses for CPU workloads. However, DREAM distinguishes itself by specifically targeting efficient memory management for GPUs within a single-node environment.

Other prior work [12, 22, 58] enable GPU direct access to the network. GPUrdma [12] is a network library for GPUs, while GPUnet [58] presents a native GPU networking layer with socket abstraction and high-level networking APIs.

7 CONCLUDING REMARKS

In this paper, we address the data transfer bottleneck between the GPU and backup memory (typically located on the CPU), which constrains performance, especially for workloads with datasets that exceed the capacity of GPU memory. We propose a novel approach for allowing the GPU to manage this data transfer directly through RDMA. Since the CPU motherboard does not support establishing RDMA connections directly, we leverage a Network Interface Card (NIC) to facilitate this direct communication. With this support, we show that DREAM can substantially outperform UVM especially when the application memory footprint exceeds the available physical memory on the GPU. In future work, we plan to explore prefetching and caching the data near the GPU to relax the requirement to have memory be pinned at the host.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and constructive suggestions. We also gratefully acknowledge the CloudLab platform [16] for providing the necessary infrastructure to conduct our experiments. This research was partially supported by the U.S. National Science Foundation under grants CNS-1955650, CCF-2212426, CCF-2339317, and FuSe-2235398.

REFERENCES

[1] NVIDIA 2020. 2020. NVIDIA Tesla A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[2] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Commongraph: Graph analytics on evolving data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 133–145.

[3] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–24.

[4] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 141–150.

[5] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[6] Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024. Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1982–2009.

[7] Chen, Zheng and Zhang, Feng and Guan, JiaWei and Zhai, Jidong and Shen, Xipeng and Zhang, Huanchen and Shu, Wentong and Du, Xiaoyong. 2023. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–31.

[8] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory harvesting in {Multi-GPU} systems with hierarchical unified virtual memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 625–638.

[9] Jack Choquette. 2023. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro* 43, 3 (2023), 9–17.

[10] City of Chicago. 2023. Taxi Trips 2013-2023. https://data.cityofchicago.org/Transportation/Taxi-Trips-2013-2023-/wrvz-psew.

[11] Rogan Creswick. 2021. Using CUDA Warp-Level Primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[12] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th international Workshop on Runtime and Operating Systems for Supercomputers*. 1–8.

[13] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[14] Djenouri, Youcef and Belhadi, Asma and Srivastava, Gautam and Lin, Jerry Chun-Wei. 2023. An efficient and accurate GPU-based deep learning model for multimedia recommendation. *ACM Transactions on Multimedia Computing, Communications and Applications* 20, 2 (2023), 1–18.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[17] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. 2024. Understanding Data Movement in Tightly Coupled Heterogeneous Systems: A Case Study with the Grace Hopper Superchip. *arXiv preprint arXiv:2408.11556* (2024).

[18] 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*. 224–235.

[19] Seokjin Go, Hyunwuk Lee, Junsung Kim, Jiwon Lee, Myung Kuk Yoon, and Won Woo Ro. 2023. Early-Adaptor: An Adaptive Framework forProactive UVM Memory Management. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 248–258.

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[21] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus. *arXiv preprint arXiv:2007.09822* (2020).

[22] Khaled Hamidouche and Michael LeBeane. 2020. Gpu initiated openshmem: correct and efficient intra-kernel networking for dgpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 336–347.

[23] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[24] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. 2024. Core Graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 18–32.

[25] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[26] 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.

[27] Marcin Knap and Paweł Czarnul. 2019. Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. *The Journal of Supercomputing* 75, 11 (2019), 7625–7645.

[28] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building heterogeneous unified virtual memories (uvms) without the overhead. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–22.

[29] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[30] Michael Larabel. 2021. AMD Making Progress on HMM-based SVM Memory Manager for Open-Source Compute. https://www.phoronix.com/news/AMD-ROCm-HMM-SVM-Memory.

[31] 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*. 1–16.

[32] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. {ROLEX}: A Scalable {RDMA-oriented} Learned {Key-Value} Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[33] John DC Little and Stephen C Graves. 2008. Little's law. *Building intuition: insights from basic operations management models and principles* (2008), 81–100.

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

[35] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. 2020. EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs. *arXiv preprint arXiv:2006.06890* (2020).

[36] Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. 2024. GPUVM: GPU-driven Unified Virtual Memory. *arXiv preprint arXiv:2411.05309* (2024).

[37] NVIDIA. [n. d.]. NVIDIA GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[38] NVIDIA. [n. d.]. *Replayable Faults*. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/uvm_gpu_non_replayable_faults.c Available at: https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/uvm_gpu_non_replayable_faults.c.

[39] NVIDIA. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[40] NVIDIA. 2020. NVIDIA DGX A100 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[41] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

[42] NVIDIA. 2023. Simplifying GPU Application Development with Heterogeneous Memory Management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.

[43] NVIDIA. 2024. Improving Network Performance of HPC Systems Using NVIDIA Magnum IO, NVSHMEM, and GPUDirect Async. https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-and-gpudirect-async/.

[44] NVIDIA. 2024. NVIDIA Grace Hopper Superchip Architecture Whitepaper. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[45] NVIDIA. 2024. Open GPU Documentation. https://nvidia.github.io/open-gpu-doc/.

[46] NVIDIA. 2024. *Welcome to the cuDF documentation!* https://docs.rapids.ai/api/cudf/stable//

[47] Jihun Park, Donghun Jeong, and Jungrae Kim. 2023. Uvmmu: Hardware-offloaded page migration for heterogeneous computing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[48] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[49] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[50] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[51] Nikolay Sakharnykh. 2019. Memory management on modern gpu architectures. In *GPU Technology Conference (GTC)*.

[52] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. 2024. Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper. In *Proceedings of the 53rd International Conference on Parallel Processing*. 199–209.

[53] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: a case for software address translation on GPUs. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 596–608.

[54] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[55] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 67–75.

[56] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[57] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*. 485–498.

[58] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)* 34, 3 (2016), 1–31.

[59] Justin Sybrandt, Michael Shtutman, and Ilya Safro. 2017. Moliere: Automatic biomedical hypothesis generation system. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1633–1642.

[60] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. 2018. Generic system calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 843–856.

[61] Benjamin Wagley, Pak Markthub, James Crea, Bo Wu, and Mehmet Esat Belviranli. 2024. Exploring Page-based RDMA for Irregular GPU Workloads. A case study on NVMe-backed GNN Execution. In *Proceedings of the 16th Workshop on General Purpose Processing Using GPU*. 7–12.

[62] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 2 (2021), 1–25.

[63] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. 1–8.

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

[65] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. 2020. A quantitative evaluation of unified memory in GPUs. *The Journal of Supercomputing* 76 (2020), 2958–2985.

[66] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 395–410.

[67] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–25.

[68] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided {RDMA-Conscious} extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 15–29.