Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions

Chuanming Shao
Shanghai Jiao Tong University
Shanghai, China
cyunming@sjtu.edu.cn

Jinyang Guo
Shanghai Jiao Tong University
Shanghai, China
lazarus@sjtu.edu.cn

Pengyu Wang
Shanghai Jiao Tong University
Shanghai, China
wpybtw@sjtu.edu.cn

Jing Wang
Shanghai Jiao Tong University
Shanghai, China
jing618@sjtu.edu.cn

Chao Li
Shanghai Jiao Tong University
Shanghai, China
lichao@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Recent GPU architectures support unified virtual memory (UVM), which offers great opportunities to solve larger problems by memory oversubscription. Although some studies are concerned over the performance degradation under UVM oversubscription, the reasons behind workloads' diverse sensitivities to oversubscription is still unclear. In this work, we take the first step to select various benchmark applications and conduct rigorous experiments on their performance under different oversubscription ratios. Specifically, we take into account the variety of memory access patterns and explain applications' diverse sensitivities to oversubscription. We also consider prefetching and UVM hints, and discover their complex impact under different oversubscription ratios. Moreover, the strengths and pitfalls of UVM's multi-GPU support are discussed. We expect that this paper will provide useful experiences and insights for UVM system design.

CCS CONCEPTS

• **Computer systems organization** → *Single instruction, multiple data*; • **Hardware** → *External storage.*

KEYWORDS

GPU, unified virtual memory, oversubscription, characterization, resource sharing

ACM Reference Format:
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 International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3489525.3511691

1 INTRODUCTION

Due to their massive thread-level parallelism capacities, Graphics Processing Units (GPUs) are now playing an important role in accelerating various real-world workloads, such as machine learning, graph processing, bioinformatics, etc. The rapid evolution of GPU architectures [3] offers new GPUs with higher computing ability and larger memory, making GPUs a promising computing platform in the future. However, compared to common server machines equipped with hundreds of gigabytes or even terabytes of memory, the memory capacity on a GPU is very limited. This restricts the scale of problems that can be solved by GPUs, especially for data-intensive applications such as machine learning and graph processing. For example, although many graph processing frameworks [17, 22, 23] could accelerate graph algorithms on GPUs efficiently, they can only solve problems with relatively small datasets. Although this situation could be alleviated by carefully partitioning the dataset and swapping partitions between host and GPU dynamically [20], it also increases programming difficulties.

Unified Virtual Memory (UVM) is a technology introduced by GPU vendors on recent GPU architectures [2, 4]. As shown in Figure 1, UVM provides a single virtual address space between CPU and GPUs, allowing automatic data migration via demand paging [7]. For example, when GPUs access pages that reside in the host memory, GPU page fault occurs and the pages will be migrated to GPU memory from the host. Meanwhile, pages could also be evicted to host when GPU memory is overwhelmed.

Although UVM induces overheads like address translation and page fault handling, it has revolutionized the GPU programming model, benefiting programmers in multiple ways. Firstly, UVM simplifies GPU programming procedures and increases programmability. As shown in Figure 2, codes that use UVM are simpler in syntax and more comprehensible. It is also straightforward to modify existing codes to take advantage of UVM. Secondly, UVM supports *memory oversubscription*, giving GPU programs the ability to use a larger amount of memory than the physical memory, without worrying about the problem of memory shortage. By utilizing UVM, existing programs can now solve larger problems with minor code modification [21].

Advanced optimization techniques, mainly *prefetching* and *memory usage hints* [1], can be used to fine-tune the performance of UVM applications, mitigating the overheads caused by UVM.

ICPE '22, April 9–13, 2022, Bejing, China.

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511691

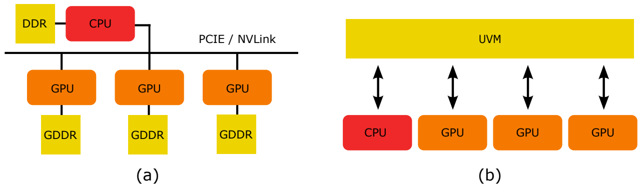


Figure 1: (a) Multi-GPU memory model, and (b) UVM programmers' view.

```
dtype *h_a, *d_a;
h_a = malloc(size);
cudaMalloc(&d_a, size);
cudaMemcpy(d_a, h_a, size);
kernel<<<...>>>(d_a);
cudaMemcpy(h_a, d_a, size);
displayResult(h_a);
```

(before)

```
dtype *a;
cudaMallocManaged(&a, size);
kernel<<<...>>>(a);
displayResult(a);
```

(after)

Figure 2: Comparison between the CUDA programs before and after using UVM programming model.

In this paper, we attempt to analyze and explain the implications of GPU UVM oversubscription. Various benchmarks are selected, and we inspect and explain the implications of oversubscription to them. After that, we consider the implications of prefetching and UVM hints under oversubscription. The possibility of utilizing UVM to support multi-GPU computation is further explored.

In this paper, we made the following key contributions:

- At the lowest level, we explore the implications of oversubscription in detail. We propose the concept of "launch ratio", and attribute it to the existence of "FALL pages". We give suggestions on reducing the adverse effects of FALL pages.
- At the system level, we investigate prefetching and UVM hints. We discover that keeping a direct mapping in the GPU page table is helpful under oversubscription ratios above launch ratio, but it is harmful otherwise. We give suggestions on choosing the suitable UVM hints.
- At the highest application level, we further discuss the multi-GPU support of UVM. We find that utilizing multiple GPUs is possible to provide good speedup in certain scenarios, but it introduces memory-related inter-GPU interferences. We give suggestions on when using multi-GPU is not appropriate.

The rest of this paper is organized as follows: Section 2 reviews the previous studies on UVM. Section 3 summarizes the experimental methods. Section 4 presents experimental results and findings. Section 5 concludes this paper.

2 RELATED WORK

UVM greatly facilitates the programmers because it reduces programming efforts and allows GPUs to process large datasets that exceed GPU memory capacity. However, the performance degradation of UVM under oversubscription should not be overlooked. Several works analyzed the characteristics of UVM and proposed techniques to improve it.

2.1 UVM Analysis

Landaverde et al. [12] first investigated UVM and noticed its significant performance overhead. Zheng et al. [25] compared the pros and cons between UVM and traditional programmer directed memory management, and find the possibility of both leverage the benefits of UVM and largely hide the performance overheads. Chien et al. [7] compared the performance of several benchmark applications with these techniques enabled and found that these the effects of these techniques are not consistent: some techniques could increase performance on one platform, but hurt performance on other platforms. Xu et al. [24] devised a machine learning-based approach to guide developers to determine the optimal memory choices, such as the combinations of various memory advice hints. Recently, Gu et al. [10] created a comprehensive benchmark suite for UVM research named UVMBench. They re-implemented a wide range of benchmarks from Rodinia [5] and Polybench [16] by employing UVM, and compared their performances with non-UVM.

2.2 UVM Management

Necessary modifications to the underlying UVM mechanisms can improve performance. For example, Ganguly et al. [8] investigated the mechanism of the tree-based neighborhood prefetcher implemented by NVIDIA CUDA driver, and proposed a tree-based pre-eviction strategy as a better replacement for the default Least Recently Used (LRU) eviction policy of NVIDIA GPUs. To mitigate the overhead of page fault handling, Kim et al. [11] proposed a hardware solution that could increase the batch size (the number of page faults handled together). They also introduced a CPU-like thread block context switching mechanism to reduce the number of batches.

Some works mentioned that the regularity of memory access could affect the application's performance under UVM. Li et al. [14] categorizes applications into regular and irregular applications, and applies three approaches (proactive eviction, memory-aware throttling and capacity compression) differently to make UVM more efficient for both regular and irregular applications. The impact of UVM on various applications is further analyzed by Ganguly et al. [9]. They discovered that under oversubscription, the LRU policy tends to evict "hot pages" of irregular applications, and they proposed a page migration policy based on access counters.

2.3 Summary of Prior Arts

To the best of our knowledge, no previous work provides a comprehensive analysis on the implications of UVM oversubscription. Although some works [9, 14] mentioned the performance degradation of applications under oversubscription, no analysis exists to reveal the diversity in the trend of performance degradation as oversubscription aggravates progressively. The general effects of prefetching and UVM hints are discussed [7], but under oversubscription, the influences of each hint, as well as whether page faults and data migration can be handled by prefetching and UVM hints remain unclear.

Relying on multi-GPU is an attractive path to GPU performance scaling [6, 15, 18], and multi-GPU memory management overhead

Table 1: GPU Benchmarks for Evaluation

Benchmark	Short Description
2dconv	2D Convolution
2mm	2 Matrix Multiplication
3mm	3 Matrix Multiplication
atax	Matrix Transpose & Vector Multiplication
backprop	Backpropagation
bfs	Breadth First Search
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
gaussian	Gaussian Elimination
gemm	Matrix-multiply
gesummv	Scalar, Vector & Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
mvt	Matrix Vector Product & Transpose
nw	Needleman-Wunsch
syr2k	Symmetric rank-2k operations

acts as an important limitation [19]. We take the first step to investigate the possibility to utilize UVM to implement multi-GPU computation.

3 EVALUATION METHODOLOGY

3.1 Design Considerations

In order to study the impact of memory oversubscription, our first step is to investigate the characteristics of individual applications. For the selected benchmarks, we are particularly interested in their performance as the oversubscription ratio increases. To this end, we need to limit the available memory on GPU according to our needs.

3.2 Benchmark Application

As listed in Table 1, we use benchmarks provided by UVM-Bench [10]. The selected applications are typical applications in various domains such as machine learning, graph theory, linear algebra, etc. Each of the applications runs a specific algorithm, and most of them only run one or two kernels during execution. The simplicity and representativeness of these applications make them suitable for our research.

3.3 Platform and Profiling Tools

The experiments are performed on a server consisting of an Intel Xeon Gold 6148 CPU with 40 physical cores, 252 GB DDR4 memory, and an NVIDIA GeForce RTX 2080 Ti card, which has 68 SMs of the Turing GPU architecture and 11 GB of GDDR6 memory. The system runs Ubuntu 16.04 OS and CUDA toolkit 11.0.

We use two NVIDIA's official profiling tools to profile the performance related data: Nsight Compute, which is used to collect the performance metrics, and Nsight System, which is used to capture and visualize the trace of events occurred during runtime.

The execution time of kernels is retrieved by inserting `cudaEventRecord` before and after each kernel invocation. We measure application's execution time by using the Linux `clock_gettime` system call, which provides a reliable real-time clock. For each application, the clock is started after all necessary preparation steps (such as data loading, memory allocation and initialization) finish, and is stopped when all kernel invocations end. The data is collected across multiple experiments and calculated by average. We also eliminate nonessential CPU-side operations such as performing the identical computations with GPU and checking the inconsistency with GPU-side results. Therefore, the measured execution time of an application is very close to the aggregated execution time of its kernels.

3.4 Memory Oversubscription Emulation

In order to understand the behavior of applications under memory oversubscription, we run them on GPU with different oversubscription ratios. To ensure consistent experimental results under different oversubscription ratios, for each benchmark application, we set a fixed workload by selecting a *fixed* dataset that is large enough (consumes about 40% of GPU physical memory) for the GPU hardware. To simulate different oversubscription ratios without changing the dataset, we first start a "memory occupier" process, which occupies a specific amount of memory on demand before the application starts. The occupied memory is allocated by `cudaMalloc`, thus this piece of memory is pinned on GPU during the application's execution.

4 RESULTS AND DISCUSSION

4.1 Sensitivity to Oversubscription

To reveal each application's sensitivity to memory oversubscription, we run applications under different oversubscription ratios and compare their execution time. *Oversubscription ratio* is the fraction of oversubscribed memory to total physical memory, which is defined as:

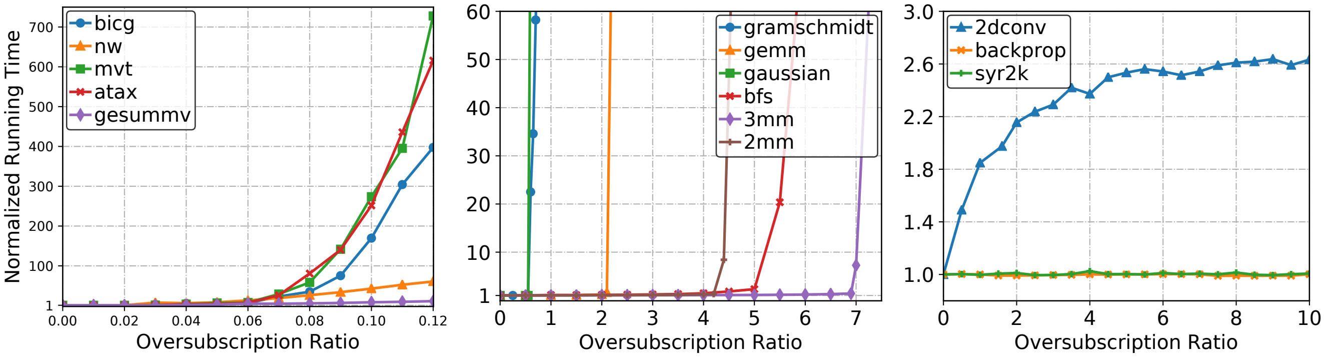
$$\text{Oversubscription Ratio} = \frac{\text{UVM} - \text{Total Physical Memory}}{\text{Total Physical Memory}}$$

As shown in Figure 3, oversubscription ratio has contrasting impact on applications. Note that in order to better observe the growing tendency of each application's execution time, we divide the applications into three groups and adjust the scales of the axes.

Applications in (a) and (b) follow an interesting behavior as oversubscription ratio increases. Specifically, the execution time keeps steady at first, but a sudden and dramatic increase in execution time occurs at some oversubscription ratio. This is especially obvious in (b), since the x-axis is more coarse-grained. This oversubscription ratio is application-specific. In this paper, we refer to this knee point as the **launch ratio** since the execution time stays constant before it but rises suddenly afterwards. For example, the launch ratio of gemm is about 2.

Applications in (c) show relative insensitivity to oversubscription. The execution time of 2dconv grows slowly and has only 2.5 times of increase when the oversubscription ratio reaches 10. Similarly, backprop and syr2k show little increase in execution time even when the oversubscription ratio reaches 10.

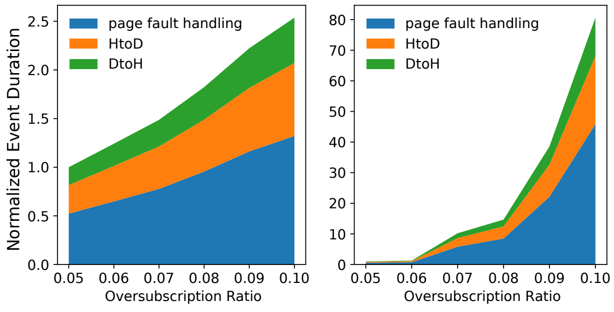
The *reason* behind these behaviors is explained in section 4.2 by analyzing the pattern of page fault. Page faults, as well as the data migrations between host and GPU, comprise the major overheads



(a) (b) (c)

Figure 3: The execution time of benchmarks with regard to different oversubscription ratios, normalized to the execution time when oversubscription ratio = 0**.**

of memory oversubscription. Figure 4 shows that as the oversubscription ratio increases, the time used for GPU page fault handling and data migration also increases. Their growing tendency is consistent with the increase of execution time in Figure 3. The time consumed by data migration is strongly correlated to page fault handling time, since each page fault induces data migration. When the physical memory in GPU is not full, a page fault simply result in data migration from host to GPU. Under oversubscription, the GPU first evict pages (move pages from GPU to host) before moving pages in from the host.



(a) nw (b) mvt

Figure 4: Comparison of the time consumed by page fault handling, host to device (HtoD) and device to host (DtoH) data migration between mvt and nw, with oversubscription ratio between 0.05 and 0.10. The time is normalized to oversubscription ratio = 0.05**.**

4.2 Pattern of Page Fault under Oversubscription

To understand the reason behind applications' behaviors mentioned in the previous section, we analyze the cause of page faults by studying the pattern of their memory access.

We select typical benchmarks and display the actual time and page numbers when page faults occur, as shown in Figure 5. It is evident that for all applications, as the oversubscription ratio increases, the amount of page faults echos with Figure 3. If the oversubscription ratio equals zero, the physical memory in GPU will be able to hold all the needed data, therefore each page is only faulted once, at the time that the page is first accessed. Under oversubscription, the size of the overall dataset is beyond the capacity of GPU memory. Thus, some pages need to be swapped between GPU and host and may fault multiple times.

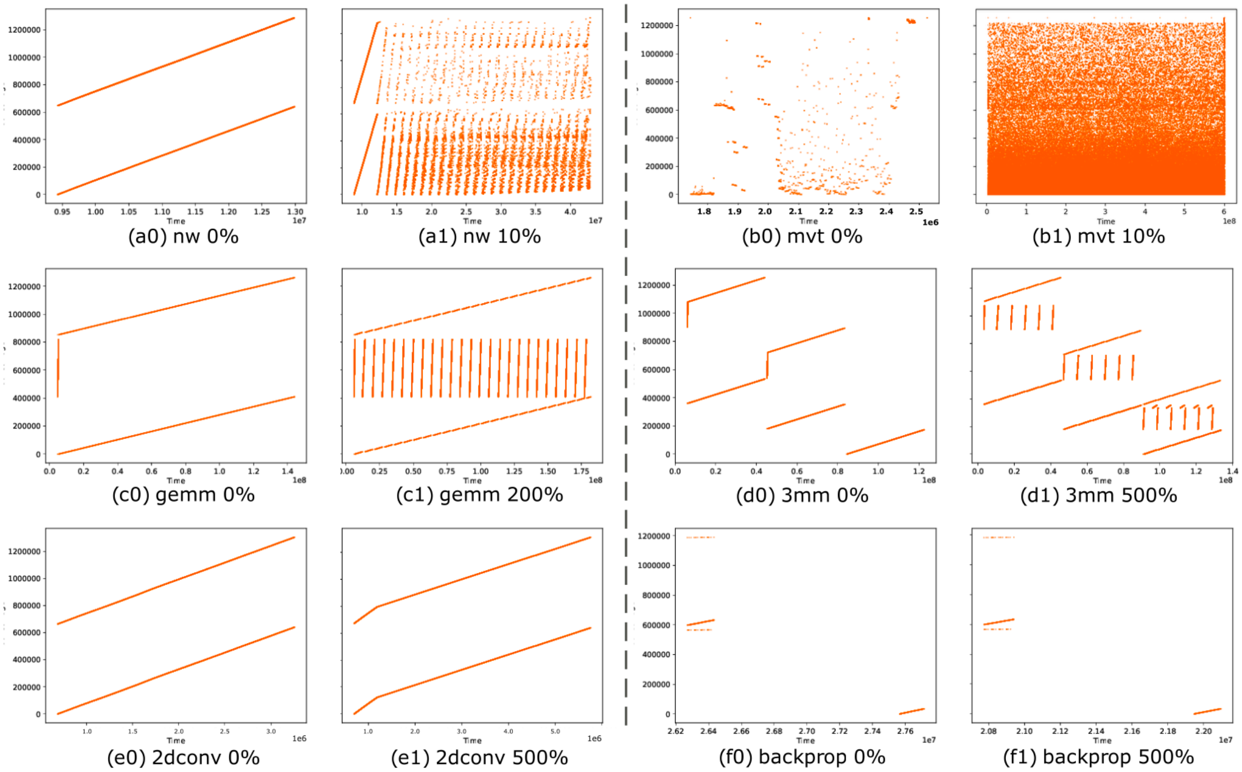
However, for different applications, the patterns of page fault have interesting diversity. For applications except mvt, we observe sequential page fault activities and line-like segments. By comparing with the source codes of these applications, we found that each segment is corresponding to a separate data structure of the application. For example, gemm is the benchmark performing matrix-matrix multiplication $C = AB$, in the manner of:

```
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++)
    for (k = 0; k < NK; k++)
      C[i][j] += A[i][k] * B[k][j];
```

gemm allocates three arrays in GPU memory: `A_gpu`, `B_gpu`, and `C_gpu`, which correspond to the matrices A, B, and C, respectively. It has three segments in Figure 5 (c0), corresponding to `A_gpu`, `B_gpu`, and `C_gpu` from low to high memory address.

As oversubscription ratio increases, some page segments may occur multiple times. For gemm, the number of page segments of matrix B grows as the oversubscription ratio increases. Although each element in `A_gpu`, `B_gpu` and `C_gpu` is accessed once, the order of accessing is different, which leads to different page fault numbers. For A and C, the elements are accessed row-wise, but B is accessed column-wise. Assuming that A, B and C are all 1024×1024 matrix stored row-wise, and each page could contain 1024 elements. To calculate one element in C, one row in A and one column in B is accessed, therefore 1 page will fault on A and C, but 1024 pages will fault on B. This means that pages of B are more likely to fault.

Pages that have similar access patterns with the `B_gpu` in gemm are harmful to applications' performance since they trigger page fault frequently, thus waste memory bandwidth. In this paper,



(a0) nw 0%

(a1) nw 10%

(b0) mvt 0%

(b1) mvt 10%

(c0) gemm 0%

(c1) gemm 200%

(d0) 3mm 0%

(d1) 3mm 500%

(e0) 2dconv 0%

(e1) 2dconv 500%

(f0) backprop 0%

(f1) backprop 500%

Figure 5: Patterns of applications' page fault under different oversubscription ratios. Each sub-figure plots the time (the x axes, in ms) when a page (the y axes) faults. Each page fault is represented by a dot, and the density of dots indicates the frequency of page faults.

we refer to the pages that are **f**requently **a**ccessed but have **l**ow **l**ocality as the **FALL pages**. The existence of FALL pages could be observed by experiments, as the page faults of FALL pages increase substantially above the launch ratio.

FALL pages are highly related to the performance of applications under oversubscription. Their proportion to GPU memory determines the application's sensitivity to oversubscription. For example, as shown in Figure 5 (a1) and (b1) , nearly all pages of nw and mvt are FALL pages, therefore they show high sensitivity under oversubscription. For gemm and 3mm, only part of their pages are FALL pages. They have steady execution time when the oversubscription ratio is low, because GPU could hold the FALL pages in physical memory. Once under some oversubscription ratio, where the physical memory is less than the size of FALL pages, nearly each memory access would lead to a page fault. Pages keep thrashing between GPU and host memory, severely hurting performance. Moreover, by referring to Figure 3 (b), we find that at launch ratio, the size of FALL pages is identical with the available memory.

The proportion of FALL pages reflects the *locality* of application's memory access. GPU physical memory could be viewed as another level in the memory hierarchy, where it acts as the "cache" of the UVM. The "hit rate" in GPU physical memory has similar impact with the cache hit rate in canonical cache hierarchy. The size of physical memory, as well as the locality of application's memory access, both contribute to an application's performance under oversubscription. Oversubscribing UVM is suited for applications that have high memory access locality such as 2dconv, but it greatly harms the performance of irregular applications such as mvt.

Naturally, *datasets* have an influence on the size of application's FALL pages. Some irregular applications have a behavior that is very data dependent. For example, the working set of graph applications such as BFS may be small even if the input graph is very large. Obviously, when running with different datasets, the size of FALL pages could change accordingly. Therefore the performance (e.g. the launch ratio) of an application can be different when different input datasets are selected.

Suggestions: Developers could reduce an application's sensitivity to oversubscription by minimizing the adverse effects of FALL pages. The first step is to locate the data structures that contain FALL pages. Often this is simple, since FALL pages are characterized by their frequent and random access. Profiling tools are also helpful in this step. After that, the developer could try to optimize the code to enhance its locality in memory access. Another straightforward option is to *pin* the FALL pages on GPU (using non-UVM).

4.3 Impact of Prefetching and UVM Hints

In this section, we study the effects of prefetching and UVM hints under memory oversubscription. Prefetching and UVM hints are the major approaches provided by CUDA, with the hope that page faults and memory thrashing could be prevented by fine-tuning the behavior of UVM at runtime. By calling `cudaMemPrefetchAsync` (PF), a memory block could be prefetched to GPU. UVM hints provide informed decisions on page handling by indicating the access patterns of data. Changing UVM hints is done by invoking `cudaMemAdvise` with one of the following policies [7]:

- `cudaMemAdviseSetAccessedBy` (AB) implies that the device keeps a direct mapping in its page table. When the data is migrated, the mapping is re-established.
- `cudaMemAdviseSetPreferredLocation` (PL) pins the data and prevents the page to be migrated, which is useful when the page is mainly accessed on one side.
- `cudaMemAdviseSetReadMostly` (RM) indicates the data region is read-intensive. It creates a read-only copy of the page on the faulting side, allowing concurrent access on both sides.

Only one policy (AB, PL, or RM) could be specified for each memory block, but each policy can be used along with prefetching. Therefore, both prefetching and UVM hints could be combined together, providing a wider design space for programmers.

In Figure 6, we compare the implications of various UVM hints to nw, gemm and backprop. Note that for each application, only one specific policy is set to all the allocated memory. In order to be better viewed, under each oversubscription ratio, the execution time is normalized to the time with UM (vanilla UVM).

It could be observed that for each application, policies have different impact on the performance. Clearly, under oversubscription, prefetching and UVM hints do not always improve the performance. Considering some memory blocks are not read-only, setting them to RM will hurt performance because once the read-only property is violated, all copies need to be invalidated, resulting in high overhead [7]. The observation that PL has little effect is likely due to our evaluation platform (Intel + PCIe) cannot take advantage of direct memory access like in NVLINK [13].

Interestingly, we observe that AB have opposite effects when running below and above the launch ratio. When oversubscription ratio is under launch ratio (0.06 for nw, 2 for gemm), the execution time with AB enabled is much longer than the execution time with vanilla UVM. However, when oversubscription ratio is above the launch ratio, AB shows to be a preferable policy. For example, when the oversubscription ratio is 0, nw runs 7 times slower with AB than with UM, but AB becomes 5 times faster than UM when the oversubscription ratio is 0.08. AB also has a similar effect on gemm. Since backprop is insensitive to oversubscription and does not have an obvious launch ratio, AB continues to hurt its performance under oversubscription.

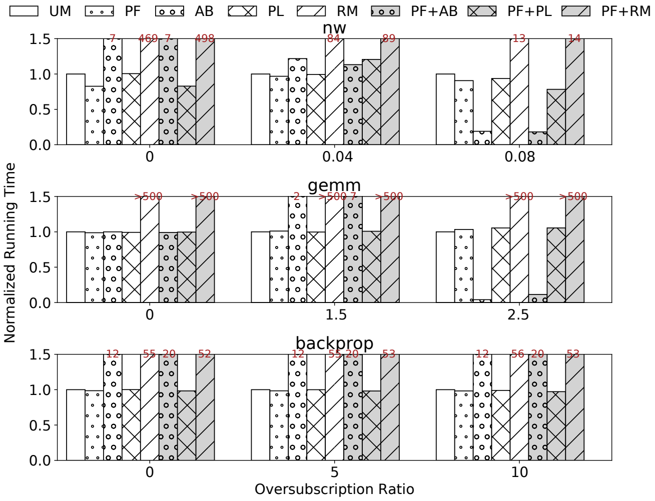


Figure 6: Implications of prefetching and UVM hints to the execution time of nw, gemm and backprop, under different oversubscription ratios. The execution time is normalized to the execution time using vanilla UVM (UM). Overly high bars are truncated and the corresponding numbers are labeled above.

The above discussions apply one hint to all data structures, which is inherently unfair, since each memory block may have different patterns of memory accessing, only one or some of the hints are appropriate. In a real-world scenario, in order to receive the optimal performance, programmers need to set different policies for each memory block, which seems to be a costly process. Therefore, a natural question is: *how large is the performance gap between the combinations of these policies?* Moreover, *does the optimal policy change under different oversubscription ratios?* To answer these questions, we investigate gemm by isolating the effects of different combinations of hints.

As previously described in section 4.2, A, B and C are the memory blocks allocated by gemm. We iterate over the combinations of policies on them, and compare their impact on the execution time. Since many policies have similar effects, we picked typical policies involving AB and RM. The execution time of gemm under different policies and oversubscription ratios are listed in Table 2. For each ratio, the policies that have relative good performance are highlighted in the table. Each policy is represented in the format of(pa, pb, pc), where pa, pb, pc are the policies of A, B, and C, respectively.

It is clear that, without oversubscription (ratio = 0), the gap between the policies is almost negligible. Under oversubscription, the policies that have good performance below launch ratio (ratio = 2) turn out to have the worst performance above launch ratio. Specifically, when oversubscription ratio = 2.5, the execution time that use policies (UM, AB, UM), (AB, AB, UM), or (AB, AB, AB) are extremely lower than other policies, while when oversubscription ratio ≤ 2, they behave much slower than the other three policies. Moreover, they have the same characteristic: they all set the policy AB to B, the array that FALL pages reside.

Overall, the effect of hint AB varies in different situations. Specifically, when oversubscription ratio is higher than the launch ratio, setting the hint of FALL pages to AB is an effective strategy, but it harms the application's performance when oversubscription

Table 2: The execution time (in seconds) of gemm under different combinations of UVM hints and oversubscription ratios

	Oversubscription Ratio					
Policy	0	0.5	1.0	1.5	2.0	2.5
(UM, UM, UM)	75.0	**77.3**	**80.1**	**83.6**	**103.6**	7911.3
(AB, UM, UM)	75.6	**77.6**	**80.0**	**84.1**	**104.6**	7982.7
(UM, AB, UM)	75.2	126.9	185.3	235.7	266.8	**288.9**
(UM, UM, AB)	75.6	**77.5**	**79.8**	**83.7**	**104.4**	8020.7
(AB, AB, UM)	75.2	127.1	186.6	233.7	267.4	**287.4**
(AB, AB, AB)	75.8	126.7	187.9	235.6	266.1	**290.1**
(RM, RM, UM)	76.4	**79.1**	**80.8**	**83.2**	**105.4**	8190.0
(RM, AB, UM)	77.1	134.4	203.1	244.4	272.1	**291.4**

is not high enough. The *reason* lies in that the hint AB could effectively reduce the thrashing. By establishing a direct mapping of the FALL pages in the page table, page faults, as well as the subsequent page migrations are both avoided. Although in this situation data must be retrived through the slower host-GPU connection, it pays off under page thrashing.

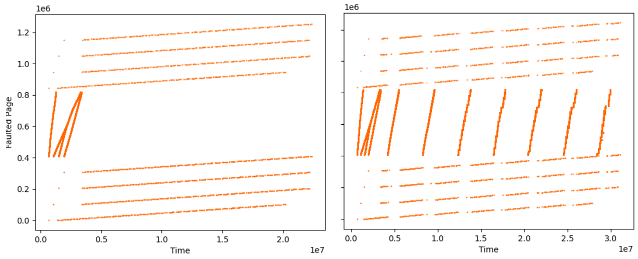
Other obvious options include using RM for *A* and/or *B*, given that they are read-only on the GPU side. As shown in the last two rows in Table 2, setting *A* and *B* to RM does not have particular advantage over other options, and has the same slowdown when oversubscription is overly high.

Suggestions: To ensure performance under all oversubscription conditions, programmer needs to choose the UVM hints dynamically based on the application's memory usage and available GPU memory. As a prerequisite, the size of the FALL pages needs to be estimated or measured by experiment. Before kernel launch, the program should first check the size of available GPU memory (e.g. via the `cudaMemGetInfo` API). If no oversubscription will happen, or the available memory is larger than the size of FALL pages, the programmer could set hints based on the conclusions provided by related researches [24]. Otherwise, based on our findings, applying the hint AB is a preferable choice.

4.4 Multi-GPU Support

The above discussions are limited to the single-GPU scenario. By allowing GPUs to access the same memory location simultaneously, multi-GPU programming is thus naturally supported by UVM. By distributing a program on multiple GPUs, the headache of oversubscription could be possibly alleviated. In this section, we demonstrate the effects of UVM under the multi-GPU setup.

It is straightforward to transform the original single-GPU program to adapt to multi-GPU. Under UVM, GPUs share the same address space, and the memory coherence when GPUs access the same memory location is ensured. Therefore by splitting a CUDA kernel function into multiple kernels, where each kernel is executed on one GPU and handles a proportion of the original job, a single-GPU problem could be transformed into a multi-GPU problem without losing correctness. To facilitate this transformation, we provide a sample programming library, available at a repository [1]. This library provides functionality to "splits" the thread blocks in a CUDA grid, and then distributes them to GPUs. We use gemm as an example.



(a) gemm 0%

(b) gemm 200%

Figure 7: The pattern of gemm's page faults, under oversubscription ratios 0 and 2.0, using 4 GPUs.

Under multi-GPU systems, RM allows each GPU to keep a duplication of the data, making GPUs to simultaneously access the data without interference. Since *A* and *B* in gemm are read-only memory, applying the hint RM on them is quite straightforward. We measure the performances of gemm under the policy (RM, RM, UM) with different number of GPUs and the result is shown in Table 3.

We could discover that using multiple GPUs provides impressive speedup. For consistency, under each oversubscription ratio, each GPU has the same amount of physical memory with the single GPU scenario. Under oversubscription ratios from 0 to 2.0, compared with the single GPU system, gemm has nearly 4x speedup when running on 4 GPUs. When the oversubscription ratio is 2.5, which is above the launch ratio, the speedup becomes lower, indicating that the inter-GPU interference become higher. We also notice that (RM, RM, AB) have severe slowdown above the launch ratio, identical as in Table 2. The slowdown also comes from the FALL pages in *B*, which is revealed in Figure 7. The pages in *B* incur more faults as oversubscription ratio increases, while the pages in *A* and *C* fault once, like in the single-GPU system.

Since the policy (RM, RM, AB) is not suitable for multi-GPU system when oversubscription ratio is overly high, we further experiment with UVM hints other than RM, for *A* and *B*. Considering that some UVM hints could be designated to GPUs separately, we explore some promising combinations of the hints in the design space. Intuitively, some UVM hints are reasonable in this scenario, including:

- Setting *B* to AB on all GPUs. Since AB eliminate page faults by keeping a direct mapping in each GPU.
- Setting *B* to PL on one GPU, and to AB on other GPUs. Since PL set the preferred location for the data, this could possibly reduce inter-GPU page thrashing.

However, based on our experiments, all of these policies bring unfavorable slowdown compared with the single-GPU system, even under no oversubscription. We identify that under multi-GPU setup, page faults caused by the FALL pages become more detrimental. For example, when using a moderate-sized dataset, under

[1] https://github.com/shawcm/GemmExample

Table 3: The execution time (in seconds) of gemm with UVM policy (RM, RM, UM) under multi-GPU setup

	Oversubscription Ratio					
# GPU	0	0.5	1.0	1.5	2.0	2.5
1	76.4	79.1	80.8	83.2	105.4	8190.0
2	39.7	40.2	41.0	42.5	49.6	6309.6
3	27.6	28.1	30.9	35.6	40.1	4276.2
4	21.0	21.1	21.3	22.1	26.4	3377.1

no oversubscription and setting *B* to AB on all *two* GPUs, 401723 host-GPU and 688434 GPU-GPU page faults occur, while only 1976 host-GPU page faults occur when it is running on *one* GPU. The page faults mainly happen when GPUs are accessing the matrix *B*, while the page faults caused by *A* and *C* are largely constant and negligible. Although PL is said to be able to *pin* the data, we do not observe this behavior on our platform (Intel + PCIe), since pages keep thrashing between GPUs when we set them to PL on one GPU. Its behavior on platforms such as NVLINK should be further studied.

Therefore, UVM is not always beneficial on multi-GPU systems. To make UVM useful, a job needs to have enough thread blocks to utilize the computation capacity of the GPUs, and have little memory interference (e.g. simultaneously access to the same page) between GPUs, or the interferenced pages are read-only.

Suggestions: Multi-GPU support is an attractive feature of UVM, and we do observe that it brings good speedup (e.g. when set RM to all read-only data of gemm). Since UVM hints do not guarantee to "pin" the pages on one GPU, using multiple GPUs seems not appropriate when FALL pages contain mutable data.

5 CONCLUSION AND FUTURE WORK

In this paper, we analyze the impact of UVM oversubscription in different aspects. 1) We observe that applications have different sensitivity to oversubscription. Further analysis shows that the memory access pattern is the major cause of the sensitivity. We find that the existence of "FALL pages" is guilty of causing the execution time to suddenly rise at the "launch ratio". 2) We inspect the effects of prefetching and UVM hints, discovering that AB is an especially significant UVM hint that could both hinder or facilitate an application's performance under oversubscription. 3) We show the speedup obtained by transforming a single-GPU program into a multi-GPU program with the help of UVM, although UVM introduces inter-GPU page faults when an application have FALL pages.

Our findings are beneficial to the design and optimization of UVM applications. The phenomenon related to "launch ratio" and the underlying reasons provide the understanding to avoid the worst-case. The effects of prefetching and UVM hints are helpful for building robust applications. We also reveal that UVM introduces new opportunities to multi-GPU computation.

Improving data locality (minimizing the number of FALL pages) is the key to avoid performance degradation under oversubscription. Relevant innovations, such as novel alogrithms, compile- and run-time optimization techniques are invaluable for improving the overall performance of UVM.

We reveal the implications of UVM oversubscription mainly by focusing on the GPU-side activities of some micro-benchmarks. However, for real-world applications, the interactions between GPU and host can be more complex, leading to various efficiency issues. Therefore, to further understand the performance-power characteristics of UVM oversubscription, more researches are required.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (No.61972247) and Shanghai S&T Committee Rising-Star Program (No.21QA1404400). We thank all the anonymous reviewers for their valuable feedback. Corresponding author is Chao Li from Shanghai Jiao Tong University.

REFERENCES

[1] [n.d.]. *CUDA C++ programming guide.* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] [n.d.]. *GCN architecture.* https://www.amd.com/en/technologies/gcn

[3] [n.d.]. List of Nvidia Graphics Processing Units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

[4] [n.d.]. *NVIDIA Pascal GPU architecture.* https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: a benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC).* 44–54.

[6] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. 2010. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS).* 1–12. https://doi.org/10.1109/IPDPS.2010.5470413

[7] Steven W. D. Chien, Ivy B. Peng, and Stefano Markidis. 2019. Performance evaluation of advanced features in CUDA unified memory. *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2019), 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014 arXiv:1910.09598

[8] 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* (Phoenix Arizona). ACM, 224–235. https://doi.org/10.1145/3307650.3322224

[9] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami 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. https://doi.org/10.1109/IPDPS47924.2020.00054

[10] 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).

[11] 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* (Lausanne Switzerland). ACM, 1357–1370. https://doi.org/10.1145/3373376.3378529

[12] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, 1–6.

[13] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[14] 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* (New York, NY, USA) *(ASPLOS '19).* Association for Computing Machinery, 49–63. https://doi.org/10.1145/3297858.3304044

[15] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU Graph Analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 479–490. https://doi.org/10.1109/IPDPS.2017.117

[16] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* 437 (2012).

[17] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: transforming irregular Graphs for GPU-Friendly Graph Processing. *ACM SIGPLAN*

Notices 53, 2 (2018), 622–636. https://doi.org/10.1145/3173162.3173180

[18] Jeff A. Stuart and John D. Owens. 2011. Multi-GPU MapReduce on GPU Clusters. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 1068–1079. https://doi.org/10.1109/IPDPS.2011.102

[19] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[20] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. *SIGPLAN Not.* 53, 1 (2018), 41–53. https://doi.org/10.1145/3200691.3178491

[21] 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.

[22] Pengyu Wang, Lu Zhang, Chao Li, and Minyi Guo. 2019. Excavating the potential of GPU for accelerating graph traversal. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 221–230.

[23] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2015. Gunrock: a High-Performance Graph Processing Library on the GPU. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* 2015-Janua (2015), 265–266. https://doi.org/10.1145/2688500.2688538 arXiv:1501.05387v6

[24] Hailu Xu, Murali Emani, Pei-Hung Lin, Liting Hu, and Chunhua Liao. 2019. Machine learning guided optimal use of GPU unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 64–70. https://doi.org/10.1109/MCHPC49590.2019.00016

[25] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357. https://doi.org/10.1109/HPCA.2016.7446077