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

Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen

Oregon State University, Corvallis, OR 97331, USA,
{guyo, wuwen, liyunf, chenliz}@oregonstate.edu

Abstract. The recent introduction of Unified Virtual Memory (UVM) in GPUs offers a new programming model that allows GPUs and CPUs to share the same virtual memory space, which shifts the complex memory management from programmers to GPU driver/ hardware and enables kernel execution even when memory is oversubscribed. Meanwhile, UVM may also incur considerable performance overhead due to tracking and data migration along with special handling of page faults and page table walk. As UVM is attracting significant attention from the research community to develop innovative solutions to these problems, in this paper, we propose a comprehensive UVM benchmark suite named *UVMBench* to facilitate future research on this important topic. The proposed *UVMBench* consists of 32 representative benchmarks from a wide range of application domains. The suite also features unified programming implementation and diverse memory access patterns across benchmarks, thus allowing thorough evaluation and comparison with current state-of-the-art. A set of experiments have been conducted on real GPUs to verify and analyze the benchmark suite behaviors under various scenarios.

Keywords: GPU, Unified Virtual Memroy, Benchmark

1 Introduction

GPUs have been gaining great attention in accelerating traditional and emerging workloads, such as machine learning, bioinformatics, electrodynamics, etc. due to GPU's massively parallel computing capability. However, there are two major issues in the mainstream GPU programming model that severely limit further utilization. First, the physical memory separation between a GPU and a CPU requires explicit memory management in conventional GPU programming model. Programmers have to explicitly copy data between CPU and GPU memories to the location where the data is used (i.e. *copy-then-execute*). Second, the conventional GPU programming model does not allow a kernel to be executed if it needs more memory that what the GPU memory can provide (i.e., *memory oversubscription*). This has greatly limited the use of GPUs in large data-intensive machine learning applications [6, 20] nowadays. Recently, GPU vendors have proposed and started to employ a new approach, *Unified Virtual Memory (UVM)*, in the newly released products [1, 16]. UVM allows GPUs and CPUs to share the same virtual memory space, and offloads memory management to the GPU driver and hardware, thus eliminating explicit copy-then-execute by the programmers. The GPU driver and underlying hardware automatically migrate the needed data to destinations. Moreover,

UVM enables GPU kernel execution while memory is oversubscribed by automatically evicting data that is no longer needed in the GPU memory to the CPU side. This is extremely important and helpful in facilitating large workloads (especially deep learning models) and GPU virtualization [8, 11] with limited memory sizes.

However, the advantages of UVM may come at a price. Analogous to virtual machines that offer great flexibility over physical machines but sacrifice performance in some degree [22], UVM also incurs performance overhead. In order to implement automatic data migration between a CPU and a GPU, the GPU driver and the GPU Memory Management Unit (MMU) have to track data access information and determine the granularity of data migration over the PCIe link [7]. This may reduce performance. For example, UVM needs special page table walk and page fault handling that introduce extra latency for memory accesses in GPUs. In addition, the fluctuated page migration granularity may also under-utilize PCIe bandwidth.

Due to the large potential benefits of UVM and its associated performance issues, UVM has recently drawn significant attention from the research community. Several optimization techniques have been proposed to mitigate the side effects of UVM [7, 9, 10, 13, 21, 23]. The earliest work is Zheng *et al.* [23], which enables on-demand GPU memory and proposes prefetching techniques to improve UVM performance. As the work predates the release of UVM, the developed on-demand memory APIs are quite different from the version in the current UVM practice. More recently, Ganguly *et al.* [7], Yu *et al.* [21] and Li *et al.* [10] study prefetching and/or eviction techniques for UVM in more detail. However, their evaluation includes only benchmarks with limited number of access patterns, which makes it difficult to assess the effectiveness of their schemes on a broader range of benchmarks with diverse memory access patterns. In fact, comprehensive benchmarks (or the lack thereof) have become a common issue in these and other prior works on GPU UVM. Most of them have used their own modified versions of existing benchmark suites (e.g., Rodinia [3,4], Parboil [17], Polybench [14]) or several in-house workloads. Our further inspection of these benchmarks shows that they lack unified implementation and no paper so far has provided a thorough analysis of the memory behaviors of these benchmarks. This can be a serious limitation for researchers and developers who aim to propose new optimizations for UVM and who would like to make comparison with existing research works.

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

Table 1: List of Benchmarks in the proposed UVMBench.

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

The developed benchmarks are evaluated on a Nvidia GTX 1080 Ti GPU with 11GB memory capacity. The code volume is reduced by removing explicit memory management APIs thanks to UVM. Evaluation results show that, if we directly implement/-convert benchmarks to the UVM programming model, there is an average of 34.2% slowdown than the non-UVM benchmarks. However, if we augment with proper *manual* optimizations on data prefetching and data reuse, the performance can be restored to almost the same as the non-UVM programming model. This indicates that there is substantial room for UVM research on developing *autonomous* memory management to close the gap between UVM and non-UVM models and possibly exceed the performance of non-UVM. Our experiment also verifies the capability of the UVM-enabled benchmarks to execute successfully under memory oversubscription scenarios, where

UVM essentially creates the illusion of a large GPU memory by using a small GPU memory and the CPU memory. While performance degradation is observed compared with a true large GPU memory, this enabling technology opens up new opportunities in accelerating large workloads on GPUs.

The main contributions of this paper are the following:

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

Table 2: UVMBench vs. other benchmarks or benchmark suites.

Benchmarks / Benchmark Suite	# of Workloads	Test in Real Hardware	ML Workloads	Diverse Memory Access Patterns	Oversubscription Support
Workloads in [7]	14	✗	✗	✗	✓
Workloads in [5]	6	✓	✗	✗	✓
Nvidia SDK [2]	1	✓	✗	✗	✗
UVMBench	**32**	✓	✓	✓	✓

We have discussed the importance of GPU UVM research and the motivation for a benchmark suite in this section. In the remaining of this paper, Section 2 describes the proposed benchmark suite in more detail. Section 3 explains our evaluation methodology. Section 4 presents and analyzes test results. Key observations drawn from the results and suggestions for future UVM research are highlighted sporadically in that section. Finally, Section 5 concludes the paper.

2 UVMBench

Benchmarks play an important role in evaluating the effectiveness and generalization when an architecture optimization is proposed. We develop a comprehensive UVM benchmark suite to facilitate the research on the GPU UVM. This suite covers a wide range of application domains marked in Table 1. The benchmarks exhibit diverse memory access patterns (more in Section 4.1) to help evaluate memory management strategies in GPU UVM. The suite also includes several auxiliary python-based programs to help create and test memory oversubscription cases. The benchmark suite is referred to as *UVMBench*, and has been made available to the GPU research community for both non-UVM and UVM versions (https://github.com/OSU-STARLAB/UVM_benchmark). Table 1 lists all the benchmarks and their configurations in UVMbench. Table 2 compares the UVMbench with some related but limited workloads in several important aspects. The development of the benchmark suite includes the following major efforts.

(1) Re-implement existing benchmarks. We start with combining three existing popular GPU benchmark suites, i.e., Rodinia [3,4], Parboil [17] and Polybench [14], removing redundant workloads and workload types, and converting into the UVM-based programming model. To implement UVM for these benchmarks, we replace all the host pointers (CPU side) and device pointers (GPU side) with a unified pointer allocated by

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

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

- *Bayesian Network (BN)* is a probabilistic-based graphical model, often used for predicting the likelihood of several possible causes given the occurrence of an event. Our implementation is based on the SJTU version [19] and, during the conversion to UVM, retains the two phases that are accelerated by the GPU: preprocessing where local scores of every possible parent set for each node are calculated, and score calculation where threads obtain the local scores and return the best one.
- *Convolutional Neural Network (CNN)* is most commonly applied to image recognition. It has also been extended to video analysis, natural language processing and many other fields. Our implementation follows the general practice where, for forward propagation, the kernels of convolutional operations, activation operations and fully connected operations are accelerated on the GPU; and for back propagation, the kernels on error calculations and weight and bias update operations are accelerated on the GPU.
- *Logistic Regression (LR)* is used to predict the probability of the existence of a certain class or event. The cost calculation is accelerated on the GPU. The input of this benchmark is the document-level sentiment polarity annotations which is first introduced in [12].
- *Support Vector Machine (SVM)* is to find support vectors that, collectively, form a hyper plane to separate different classes. In our implementation, the kernel matrix calculation is accelerated on the GPU. The code is based on the Julia project [15] and converted to UVM.

Listing 1.1 shows the partial code of the sigma update function in the SVM benchmark, which demonstrates the re-implementation process and newly added benchmarks. Several unrelated variables are omitted for simplicity. In the traditional programming model, due to explicit memory management, the program without UVM would need to allocate memory space on the device by using a number of *CudaMalloc* and *CudaMemcpy* APIs before and after a kernel launch to explicitly migrate the required data between the host and the device. In contrast, the UVM programming model shown in Listing 1.1 unifies the memory space of the host and the device. By calling *cudaMallocManaged* APIs (lines 6-7), the code allocates bytes of managed memory. The allocated variables can be accessed by the host and the device directly, and are managed by the Unified Memory system of the GPU. When this Sigma_update function is called in the main function (line 1), the variables, defined by *cudaMallocManaged*, are passed into the function, and the device kennels can directly access these variables. Therefore, the

UVM programming model greatly reduces the code complexity by removing device variable definitions and memory management APIs.

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

Listing 1.1: *Sigma_Update* function in SVM with UVM.

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

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

Benchmarks in the proposed UVMBench are all implemented in CUDA and can be run on Nvidia GPUs. This suite includes both the non-UVM version (original) and the UVM version implementation for performance comparison. There are no algorithmic changes when developing the UVM version of the benchmarks. This ensures fair comparison between the traditional programming model and the UVM programming model. Consequently, the observed performance changes are mostly attributed to the difference between programming models rather the algorithms.

Some previous works [5,7] and the Nvidia SDK present a limited number of UVM-enabled workloads to demonstrate the effectiveness of the UVM or their proposed ideas. Table 2 compares the existing benchmarks with our proposed UVMbench in five important aspects. Compared with the existing benchmarks, UVMbench presents more workloads from different domains. In particular, UVMbench includes machine learning workloads to explore the possibility of applying UVM techniques in data-intensive

machine learning applications. Moreover, UVMbench provides diverse memory access patterns and supports memory oversubscription.

3 Evaluation Methodology

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

To conduct the above experiments, we employ an Nvidia GTX 1080 Ti GPU with the Pascal architecture. We use the Nvidia Binary Instrumentation Tool (NVBit) [18] to extract the global memory access patterns of the UVMBench suite. NVBit provides a fast, dynamic and portable binary instrumentation framework that allows users to inspect/instrument instructions. we use two Nvidia official profiling tools to profile the performance related data of benchmarks: nvprof, a command line tool to collect and view profiling data, and Nvidia Visual Profiler, a GUI to visualize the application performance.

4 Results and Analysis

4.1 Memory Access Pattern Profiling

To study the relationship between memory behaviors and UVM efficiency, we first profile memory access patterns of each benchmark. In this experiment, NVBit is used to generate memory reference traces by injecting the instrumentation function before performing each global load/store. The memory traces are plotted in Figure 1. The horizontal axis corresponds to the logical access time, and the vertical axis shows the accessed memory addresses.

As can be seen from the figure, benchmarks in the UVMBench suite exhibit diverse memory access patterns. They can be generally classified into *regular* and *irregular* memory access patterns, as indicated after each benchmark name as (R) or (I) in Figure 1 (and as indicated in the "Type" column in Table 1). This classification follows the same classification method as [10]: if benchmarks access only a small number of memory pages at any point of time, they are classified as *regular* benchmarks; in contrast, benchmarks with large unique memory pages access at a given time are identified as *irregular* benchmarks. For regular benchmarks (e.g., 2DCONV, 2MM and so on), they exhibit a streaming access pattern. These benchmarks access only a small number of memory addresses and seldom exhibit data reuse within the kernel. In contrast, irregular benchmarks show very different memory access patterns: accessing many memory addresses at a given time (e.g., ATAX, BICG, GAUSSIAN), repeatedly accessing the same memory address over time (e.g., COVAR, GRAMSCHM), or accessing random addresses (e.g., SC, SVM). Note that benchmark *NW* is classified as irregular, as it exhibits a sparse, localized and repeated memory accesses, although this is not quite visible in the figure due to the scale. In the experiment of memory oversubscription

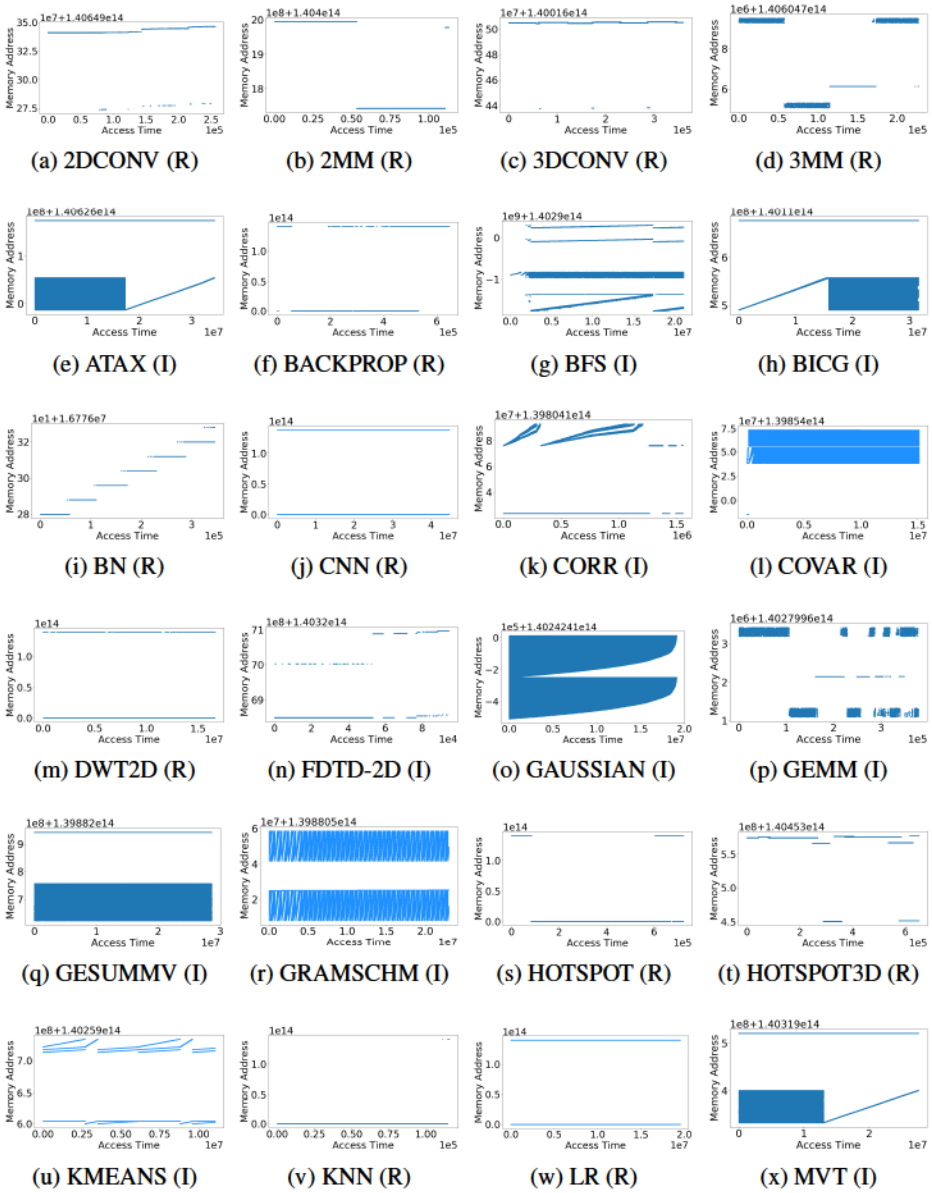
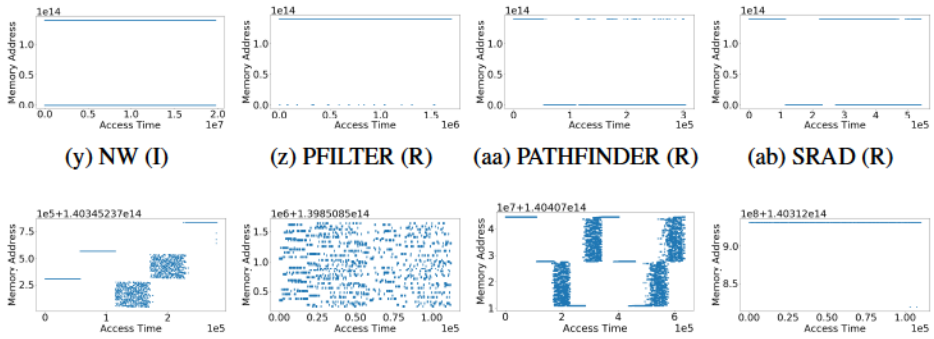


Fig. 1: Memory access patterns of benchmarks in UVMBench.



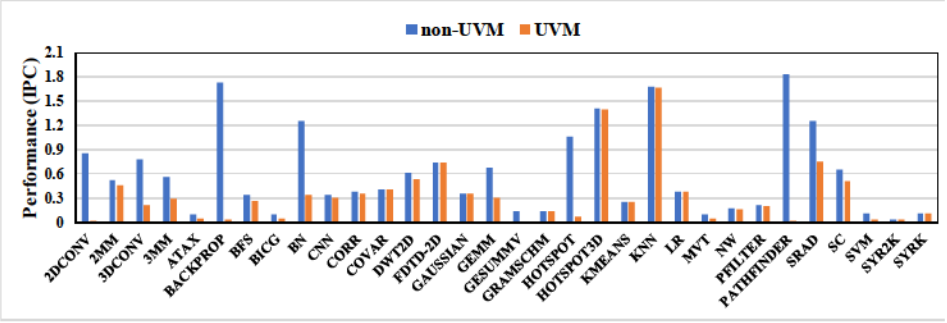


Fig. 2: Direct UVM conversion in UVMBench leads to large performance degradation vs. non-UVM.

presented later in Section 4.4, we find that benchmark performance is highly related to memory access patterns.

4.2 UVM *vs.* non-UVM Performance

a. Performance of Direct UVM Conversion

As mentioned earlier, while UVM greatly eases programming efforts by removing explicit memory management, this is achieved at the cost of certain performance overhead, particularly with naive/direct conversion to UVM. Figure 2 compares the performance of all the benchmarks in the non-UVM and UVM programming models. The IPCs are obtained from Nvidia *nvprof*. Across the benchmarks, the performance of the UVM version has an average of 34.2% slowdown compared with the non-UVM one. These results are expected as the page fault handling causes large performance overhead for kernel execution. Under the UVM programming model, data is allowed to reside in other location (e.g., on the CPU side) while a kernel is executing. When the required data does not reside in the GPU DRAM (page fault occurrence), the kernel has to be stalled while waiting for the data to be fetched from the CPU side. In the non-UVM version, programmers have made sure that data is always available on the GPU side.

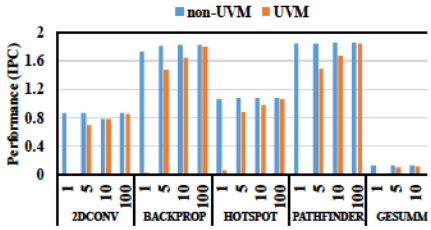


Fig. 3: Performance of UVM restores wi increased number of kernel invocations.

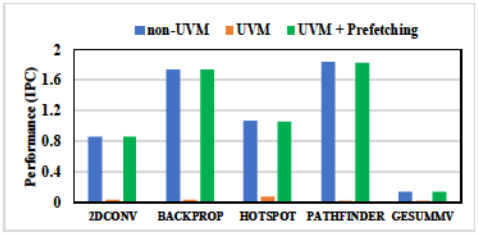


Fig. 4: Performance of UVM restores by enabling prefetching.

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

b. Restoring UVM Performance via Data Reuse

Data reuse can mitigate UVM performance degradation by reducing the occurrence of page faults. As mentioned earlier, we study the impact of data reuse by modifying the number of times a kernel is invoked. Figure 3 plots the change in performance as we increase the kernel invocation times (there is no kernel execution dependency between consecutively invoked kernels). It can be seen that the performance of these benchmarks under UVM is rapidly improving with more invocation and eventually approaches to the performance of non-UVM. Except for the first executed kernel, the following kernels in the GPU program may reuse the data that has been fetched during the execution of the first kernel, and fewer page faults would occur. The results confirm that more data reuse leads to smaller data migration overhead.

Observation/Suggestion: Although data reuse is artificially introduced in the software program in this experiment, it prompts us that if applications exhibit significant data reuse opportunities, either inherent or created through architecture optimizations, UVM can be an attractive model that provides flexibility while having little performance overhead.

c. Restoring UVM Performance via Data Prefetch

Nvidia provides a runtime API *cudaMemPrefetchAsync* that enables asynchronous data prefetching. Through this API, data can be prefetched to the device memory before the data is accessed by a kernel on the GPU. This reduces the occurrence of page faults. To study the impact of prefetching on UVM kernel execution performance, we augment all the benchmarks in UVMBench with such prefetching capability. Figure 4 shows the results from the above 5 benchmarks that experience the largest performance drop in UVM.

It can be observed that the performance of these benchmarks improves considerably after this optimization and is close to the performance of the non-UVM version. The

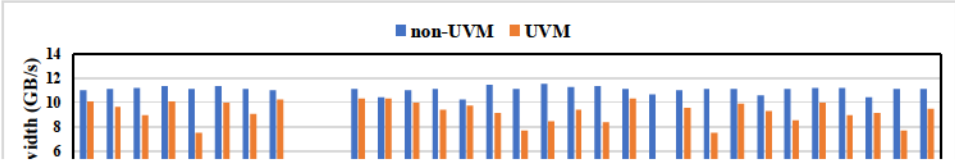
geometric mean of the slowdown has decreased from 95.8% to merely 0.7%. The improvement comes from the fact that kernel execution is now rarely stalled as data has already been fetched in the device memory before being accessed. While not shown, the performance of other 27 UVM-version of the benchmarks also restores to very close to the non-UVM version after using asynchronous prefetching.
Observation/Suggestion: Besides data resue, another alternative to restore performance degradation of UVm is data prefetching by employing the runtime API *cudaAsyncPrefetch*. In theory, page faults can be completely eliminated if there is an oracle prefetecher that is able to load any required data into the GPU memory before the data is accessed. That can serve as an upper-bound of future UVM prefetech schemes.

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

4.3 Effect of Data Migration on PCIe Bandwidth

The performance of data migration between CPU and GPU also closely relates to the effective PCIe bandwidth. Under the UVM programming model, variable sized on-demand data is transferred from the CPU memory to the device memory. To understand performance trade-offs, it is worth studying the effect of UVM data migration on the PCIe link. Figure 5 compares the achieved PCIe bandwidth with non-UVM and UVM programming models during data migration. On average, the achieved PCIe bandwidth of UVM is 15.2% lower than that of non-UVM. In general, the larger the transferred data size is, the higher the effective PCIe bandwidth can achieve. This is mainly because of the constant PCIe protocol overhead and limited hardware resources (e.g., data buffer size, number of DMA channels, number of outstanding requests, etc.), so the overhead can be amortized better with larger transferred data. Since the non-UVM model copies the entire allocated data chunk to the GPU memory before execution, this results in relatively high effective bandwidth. In contrast, the migrated data size in UVM is usually much smaller than the non-UVM one as only on-demand data is migrated through the PCIe bus (usually smaller than 1MB). Note that benchmarks *BN* and *CNN* in UVM and non-UVM both exhibit low effective PCIe bandwidth, because the sizes of allocated variables in these two benchmarks are all small (less than 4KB), and even the entire chunk of allocated variable transmission cannot fully utilize the PCIe bandwidth.

Figure 5 also shows that, among UVM benchmarks, the effective PCIe bandwidth may vary a lot. The variation is mainly caused by the hardware prefetcher inside the GPU. For example, Nvidia has implemented a tree-based hardware prefetcher in their GPUs, which heuristically adjusts the prefetching granularity based on access locality. The difference in memory access patterns across benchmarks put the hardware prefetcher in different degrees of efficacy. More detailed discussion on UVM hardware prefetchers can be found in other papers such as [7, 10, 21].



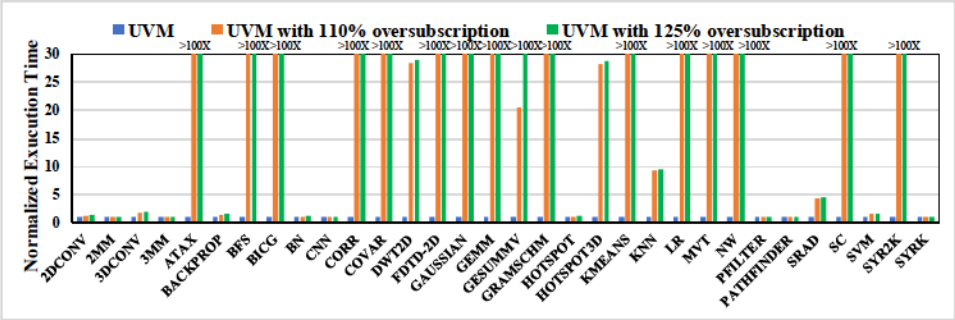


Fig. 6: Change in benchmark execution time when GPU memory oversubscripted (normalized to no memory oversubscription).

Observation/Suggestion: The above results on the effective PCIe bandwidth indicate that hardware prefetchers that are currently employed in GPUs cannot fully utilize PCIe bandwidth. Thus, future research is much needed to continue developing and optimizing GPU hardware prefetchers that are UVM-aware.

4.4 Oversubscription

A major advantage of UVM is to enable kernel execution when memory is oversubscribed. Performance under memory oversubscription can be significantly reduced since part of the data now needs to be brought from the CPU memory. Despite this, UVM is still very attractive, as such memory oversubscription is not possible under non-UVM. To quantify the performance degradation when the GPU memory is oversubscribed, we run all the benchmarks in the suite under various memory capacities. As different benchmarks have different required memory footprint, to create memory oversubscription, we modify the available memory space through the *cudaMalloc* runtime API. The required memory footprint is set to be 110% and 125% of the available memory space in the GPU physical memory. Figure 6 shows the results. As expected, all the benchmarks suffer considerable performance degradation under memory oversubscription. The more memory is oversubscribed, the more performance degrades.

From Figure 6, we also observe that many of the benchmarks can complete execution with 2-3x slowdown under memory oversubscription, whereas other benchmarks suffer from a significant performance penalty or even crash, marked as >100X in the

figure (e.g., LR uses the cublas library which cannot support memory oversubscription and leads to crash). For the former, we find that the main performance overhead is caused by kernel stalls when waiting for the eviction of pages to create space for newly fetched data. These benchmarks usually have a streaming access pattern (Section 4.1). With this pattern and the LRU eviction policy in Nvidia GPUs, the evicted data does not affect kernel execution as the evicted data is not reused any more. Therefore, the performance overhead mainly comes from the waiting time of page eviction. For the latter, the large performance penalty mainly comes from severe page thrashings, which repeatedly migrate the page back and forth between the GPU and the CPU. This usually occurs when a benchmark has a short data reuse distance so the evicted data is needed/reused within a short time. Note that, although the degradation seems large, UVM is still much better non-UVM which does not allow kernels to run at all if the memory is oversubscribed.

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

5 Conclusion

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

Acknowledgement

We sincerely thank the reviewers for their helpful comments and suggestions. This research was supported, in part, by the National Science Foundation (NSF) grant #1750047.

References

1. Radeons next-generation vega architecture, 2017. https://radeon.com/downloads/vega-whitepaper-11.6.17.pdf.
2. Cuda toolkit, 2020. https://developer.nvidia.com/cuda-downloads.
3. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization*, pages 44–54, 2009.
4. S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization*, pages 1–11, 2010.
5. S. Chien, I. Peng, and S. Markidis. Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019.

6. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
7. D. Ganguly, Z. Zhang, J. Yang, and R. Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 224–235, 2019.
8. M. Gu, Y. Park, Y. Kim, and S. Park. Low-overhead dynamic sharing of graphics memory space in gpu virtualization environments. *Cluster Computing*, pages 1–12, 2019.
9. H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.
10. C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63, 2019.
11. Q. Lu, J. Yao, H. Guan, and P. Gao. gqos: A qos-oriented gpu virtualization with adaptive capacity sharing. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):843–855, 2019.
12. A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.
13. P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426. IEEE, 2018.
14. L.-N. Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.
15. Y. Qin. a stochastic decomposition implementation of support-vector machine training. https://github.com/qin-yu/julia-svm-gpu-cuda, 2019.
16. N. Sakharnykh. Unified memory on pascal and volta, May 2017.
17. J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.
18. O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.
19. Y. Wang, W. Qian, S. Zhang, X. Liang, and B. Yuan. A learning algorithm for bayesian networks and its efficient implementation on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):17–30, 2015.
20. W. Wu, Z. Qi, and L. Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9621–9630, 2019.
21. Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang. A quantitative evaluation of unified memory in gpus. *The Journal of Supercomputing*, pages 1–28, 2019.
22. W. Zhang, M. Zhu, T. Gong, L. Xiao, L. Ruan, Y. Mei, Y. Sun, and X. Ji. Performance degradation-aware virtual machine live migration in virtualized servers. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 429–435, 2012.
23. T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.