2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00063

PILOT: a Runtime System to Manage Multi-tenant GPU Unified Memory Footprint

John Ravi, Tri Nguyen, Huiyang Zhou, Michela Becchi
NC State University
{jjravi,tmnguye7,hzhou, mbecchi}@ncsu.edu

***Abstract*—Concurrent kernel execution on GPU has proven an effective technique to improve system throughput by maximizing the resource utilization. In order to increase programmability and meet the increasing memory requirements of data-intensive applications, current GPUs support Unified Virtual Memory (UVM), which provides a virtual memory abstraction with demand paging. By allowing applications to oversubscribe GPU memory, UVM provides increased opportunities to share GPU resources across applications. However, in the presence of applications with competing memory requirements, GPU sharing can lead to performance degradation due to thrashing. NVIDIA's Multiple Process Service (MPS) offers the capability to space share bare metal GPUs, thereby enabling cluster workload managers, such as Slurm, to share a single GPU across MPI ranks with limited control over resource partitioning. However, it is not possible to preempt, schedule, or throttle a running GPU process through MPS. These features would enable new OS-managed scheduling policies to be implemented for GPU kernels to dynamically handle resource contention and offer consistent performance.**

The contribution of this paper is two-fold. We first show how memory oversubscription can impact the performance of concurrent GPU applications. Then, we propose three methods to transparently mitigate memory interference through kernel preemption and scheduling policies. To implement our policies, we develop our own runtime system (PILOT) to serve as an alternative to NVIDIA's MPS. In the presence of memory oversubscription, we noticed a dramatic improvement in the overall throughput when using our scheduling policies and runtime hints.

I. Introduction

In order to facilitate programming and meet the increasing memory requirements of data-intensive workloads, NVIDIA GPUs offer Unified Virtual Memory (UVM) support to applications through CUDA which is a general purpose computing API for NVIDIA's GPU. This feature provides a virtual address space encompassing the physical memories of the CPU and the GPUs on a compute node, as well as full virtual memory support with demand paging since CUDA 8. Thanks to demand paging and migration, programmers no longer need to manually partition the application's data set to fit it into the available GPU memory, and explicitly stage memory transfers between CPU and GPU. In addition, by relaxing memory utilization constraints, this feature provides more opportunities for sharing a GPU among processes and applications. This can be particularly beneficial for iterative applications, with each kernel invocation utilizing only a fraction of the application's data set. While in its early stages UVM had a non negligible performance cost [1], UVM support has been progressively improved, and nowadays on data sets fitting the GPU memory capacity UVM offers performance comparable to explicit data transfers between CPU and GPU, especially for applications exhibiting regular memory access patterns [2]. However, GPU memory oversubscription comes at a performance cost, in some cases significant, even for kernels and applications running in isolation on a GPU [3]–[6]. This problem can be exacerbated in the presence of inter-application concurrency. Indeed, our experiments show that a kernel can drastically hinder performance of co-running kernels by triggering a large number of page faults.

The goal of this work is to address memory oversubscription issues in the presence of concurrency on current GPU platforms. In particular, we aim to provide a software runtime solution that allows multi-tenant kernel execution across workload processes while mitigating performance issues that affect quality of service. In summary, this work makes the following contributions:

- We propose a runtime system, called PILOT, that enables GPU hardware space sharing with UVM.
- We demonstrate concurrency problems that can result from the oversubscription of device memory and propose three scheduling policies aimed to share the GPU hardware resources while mitigating performance degradation due to device memory oversubscription. We incorporate these mitigation schemes in PILOT. The first two schemes proposed, *MFit* and *AMFit*, require kernel preemption support while the third one, *MAdvise*, does not.
- We validate PILOT on applications from the Rodinia [7] and PolyBench-GPU [8] benchmark suites using a Pascal and a Volta GPU, and compare it with Nvidia MPS.

II. Background and Motivation

A. The Case for GPU Multi-Tenancy

NVIDIA's Multi-Process Service (MPS) is a runtime system that enables multiple CPU processes to run kernels concurrently on a single GPU. MPS has been designed to enable multi-process applications (e.g., MPI jobs) to share a GPU. When running, this service transparently intercepts any CUDA calls issued to the CUDA runtime. This avoids the need for switching CUDA contexts when running multiple processes (time sharing), and it allows processes to share GPU data without performing IPC (space sharing).

Fig. 1 shows the performance benefit of MPS when multiple processes issue work to a single GPU (Titan Xp for Pascal

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00063

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.

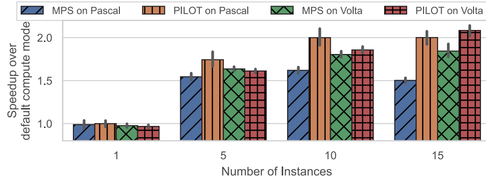


Fig. 1: Speedup offered by MPS and PILOT over the bare CUDA runtime in the absence of memory oversubscription (y-axis). We run a varying number of concurrent *VectorAdd* programs (x-axis). Each *VectorAdd* instance has a 512MB device memory footprint.

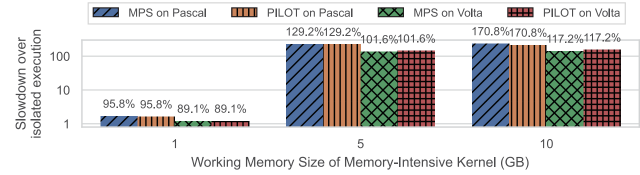


Fig. 2: Slowdown of *VectorAdd* kernel when co-run with a memory-intensive kernel over when running in isolation (y-axis). Along the x-axis we vary the memory footprint of the memory-intensive kernel. The memory subscription percentage is indicated on top of each bar. In these experiments PILOT mitigations are not enabled.

architecture or Tesla v100 for Volta architecture) concurrently. We run multiple instances of a simple *VectorAdd* CUDA program, each instance allocating about 0.5 GB of memory using UVM. We perform three sets of experiments: one using MPS, one using our PILOT runtime, and one using the CUDA runtime alone (baseline). This figure also plots the runtime variance in the form of confidence intervals. By enabling multiple processes to space share the GPU, in the presence of multiple *VectorAdd* instances MPS and PILOT lead to a noticeable speedup.

B. GPU Concurrency QoS Limitations

Volta GPUs enable enhanced MPS capabilities, such as allowing processes to submit their work directly to the GPU (rather than passing through the MPS server). Second, it provides process isolation (i.e., each process has an own GPU address space) and limited provisioning of GPU resources [9]. Volta MPS aims to implicitly throttle hardware resources by capping the number of logical resources (threads) for each CUDA context; however, it does not directly limit access to physical resources such as memory and execution units.

On both the Pascal and Volta architectures, MPS does not offer quality of service (QoS) guarantees. While often beneficial, concurrent kernel execution can be prone to problems, such as one process adversely affecting the performance of other processes. Previous work [10], [11] has focused primarily on contention on on-chip resources, such as registers and shared memory. Here, we focus on device memory contention, especially in the presence of UVM. With UVM, a memory-intensive kernel causing a lot of page faults can dramatically affect the performance of all other kernels sharing the GPU.

To test this behavior, we designed a memory-intensive kernel that runs indefinitely and touches as many pages as it can in a short amount of time. The specifics of this kernel are described in detail in Section IV-A. Fig. 2 shows performance results obtained when varying the memory footprint of the memory-intensive kernel from 1GB to 5GB to 10GB, and running it along with an instance of *VectorAdd* with a working memory size of 512MB. The experiments are performed on a Titan Xp GPU equipped with a 12GB device memory and a Tesla v100 equipped with 32GB device memory. To demonstrate oversubscription we reserve a fixed amount of GPU memory using a *cudaMalloc* and *cudaMemset*. By reserving 85% of the available 12GB device memory on the Titan Xp and 32GB device memory on the Tesla v100, we were able to achieve oversubscription with smaller working memory sizes of our memory-intensive application and *VectorAdd* application. With a memory-intensive kernel allocation of 5GB, the device memory is more than 100% subscribed on both architectures. Thus, when the memory footprint of the memory-intensive kernel increases beyond 5GB, there is a noticeable slowdown in kernel execution time of *VectorAdd* under NVIDIA MPS on Pascal, NVIDIA MPS on Volta, and PILOT (with memory interference mitigation disabled).

III. PILOT'S DESIGN AND IMPLEMENTATION

To evaluate our memory management techniques, we implemented a GPU runtime system, called PILOT. PILOT enables concurrent kernel execution, provides UVM support, and incorporates mechanisms to manage memory oversubscription among GPU applications. PILOT extends our previously proposed GPU virtualization system [10], [12] with UVM and memory-aware scheduling support. Fig. 3 shows PILOT's high level architecture. PILOT includes two components: a front-end library and a back-end daemon. The front-end library (called $libcudart.so$) is shared and defines all the function calls that PILOT intercepts; the back-end daemon is a proxy process sitting between the running applications and the CUDA runtime. CUDA API calls interception is done by utilizing the `LD_PRELOAD` environment variable to override calls to the CUDA runtime. At the start of each program, the front-end establishes a connection to the back-end daemon through UNIX sockets, and it then uses this connection to redirect CUDA function calls to the back-end. The back-end leverages the use of multiple CPU threads to concurrently execute GPU applications. Specifically, it associates to each GPU a set of CPU threads - called vGPUs - whose number is configurable. vGPUs act as proxy threads: they issue to the CUDA runtime function calls that have been intercepted by the front-end and redirected to the back-end for scheduling and dispatching. PILOT leverages CUDA streams [13] to concurrently execute each program, and it associates a non-blocking stream to each vGPU. This allows for a configurable amount of parallelism. Mapping of GPU work to vGPUs is performed by a *Scheduling Thread*, which implements the scheduling policies described in Section III-C.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.

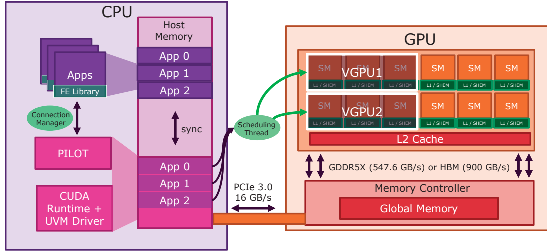


Fig. 3: High-Level Design of the PILOT Runtime Framework

A. *UVM Support*

We note that, in our design, the PILOT back-end is the only process exposed to the CUDA runtime. As a consequence, the address spaces of the applications are decoupled from the address space seen by the CUDA runtime (and the GPUs). This, in turn, complicates the support of CUDA programs utilizing UVM. In order to support UVM, PILOT must handle the page tables across the CUDA applications' and the back-end process's address spaces. With CPU memory allocations, it is possible to share virtual memory references by using *mmap* to create a shared memory. However, we have no control over the implementation of the CUDA memory allocator. Since each memory address space is tied to a CUDA context, and there is no CUDA API functionality to share a CUDA context or its memory mappings between the runtime process and the application processes, we need to maintain a separate copy of the page tables content which needs to be synced at certain times during execution. To accomplish this, we incorporate in PILOT a functionality proposed by Garg et al. [14] to provide checkpoint-restart support for GPU applications. Specifically, they proposed using a *shadow region table* (a software-managed page table that maps one or more pages from one address space to another) to checkpoint UVM-managed data and share memory pages with a proxy process. Their approach utilizes the Linux segfault handler to track which address space contains the "latest" copy of the data.

We designed the PILOT daemon to run on the same system as the programs that it intercepts. This allows us to perform more efficient data copies by using two Linux system calls: *process_vm_read* and *process_vm_write*. These system calls allow transfering data between the address spaces of the intercepted application and the PILOT daemon without any intervention from their respective remote process. Furthermore, the data move directly between the address spaces of the two processes without needing to pass through the kernel space.

B. *Kernel Preemption*

Kernel preemption allows implementation of more complex scheduling policies at the software level. However, CUDA (as of version 11.4) does not offer API functions to preempt a running kernel short of killing the host process that made the CUDA call. CUDA API calls such as *cuStreamDestroy*, *cuCtxDestroy* and *cudaDeviceReset* result in a blocking stall that causes applications to wait for the running kernel to finish its execution. While GPU architectures since Pascal support instruction-level preemption, this feature has not been exposed through the CUDA runtime or driver API.

TABLE I: Overhead of coarse-grain software preemption using the small, medium and large datasets of Table II. The data indicate the slowdown over preemption-free execution.

Benchmark kernel	small	med.	large
backprop	4.36	1.43	1.17
bfs	2.02	1.44	1.40
hs3d	1.70	1.69	1.66
partfilter	3.75	3.66	2.94
lavamd	1.33	1.01	1.00
nn	2.23	1.26	1.22
gaussian	5.86	3.09	2.40
corr	1.53	1.14	1.08
gesummv	2.64	1.94	1.17
2dconvol	6.85	1.70	1.08
geomean	2.55	1.56	1.33

To achieve software preemption of kernels without manual intervention, we leverage a dynamic instrumentation tool [15] to dynamically inject code into CUDA kernels. We use a global flag set by the PILOT runtime to indicate if the kernel should stop running and inject in the kernel some code to check the value of this flag and, if set, preempt the kernel's execution. We propose a coarse-grain preemption scheme which aims to reduce preemption overhead for long running kernels. Typically, long-running kernels include iterating loops. Thus, the coarse-grain approach inserts preemption checks before backward branches inside the kernel code. Since checking at every back-edge inside a deeply nested loop might still be too expensive, we only insert preemption-checks before the back-edge of the outer-most loop inside the kernel.

Kernel preemption introduces two overheads: instrumentation and runtime overhead. The former can be ignored, since the kernel code can be instrumented one time before running the application. The runtime overhead depends on two factors: the number of instructions injected in the kernel and the location of those instructions. Table I shows the slowdown in kernel execution caused by coarse-grain preemption when utilizing a quarter of available SMs on an NVIDIA v100 GPU. As can be seen, our coarse-grain method introduces a limited overhead, which tends to decrease as the data set size and the kernel runtime increase.

Kernel preemption is complicated by the need to checkpoint-recover intermediate data which can introduce additional latency. We observe that this overhead can be avoided in certain cases. If we can determine that an application does not modify any input data and has no side effects, our runtime does not need to recover any input data after preemption. In addition, some convergence-based applications can be tolerant to interruptions even if they modify their input data. For example, in BFS each kernel iteration assigns each node at a level with its depth value. If that kernel iteration were interrupted and only a subset of the nodes were assigned a depth value, recomputing the depths will still result in the same answer since it will override any values from the previous iteration. In our experiments, when necessary, we modified the benchmarks to avoid the need for checkpoint-recovery.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.

C. Mitigation Schemes

Here, we discuss the three mitigation schemes that we designed and implemented in PILOT to address the memory interference problem of concurrent GPU kernels.

1) Memory Fit (MFit): Our first scheme is a subscription-based scheduler called Memory Fit which aims to limit the oversubscription of the GPU memory by only scheduling a kernel when there is enough free memory on GPU to satisfy the kernel's estimated memory footprint. To achieve this, the Scheduling Thread polls the GPU memory usage and only invokes a kernel when there is enough space on the GPU to accommodate all of its allocated memory buffers. If there are no concurrent kernels running, then the kernel launch request will be serviced immediately. To prevent starvation, each application is associated a user-defined maximum wait time. When this time is exceeded, PILOT preempts running kernels (in descending order of memory allocated size) until enough memory is freed. This approach has two limitations. First, it does not support pointer-based data structures (such as linked lists or nested structures with pointer fields). It is worth noting that pointer-based data structures are rarely used in CUDA applications, and they are not included in any of the benchmarks used in our experiments. Second, demand paging (supported on Pascal and later GPUs) allows CUDA kernels to bring data in device memory only when required. Thus, the memory usage estimate using this technique can significantly exceed the actual memory footprint of the kernel and be too conservative.

2) Active Memory Fit (AMFit): The shortcoming of the MFit scheme is that an application might allocate more memory than it might actively use in a particular kernel launch. Our Active Memory Fit scheme addresses this issue by monitoring the active memory usage of each CUDA kernel. Since we do not have direct access to the GPU driver information, we identify and track the memory allocations made with the CUDA UVM driver by monitoring the CPU page table. We found that, when a page is sent to the GPU, it is marked as clean in the CPU page table. So, our runtime forces all CPU pages allocated by CUDA UVM to be marked as dirty before any CUDA kernel invocation. This way, we can tell exactly how many pages are used by a CUDA kernel at runtime. When a CUDA kernel request is sent to PILOT, the kernel is launched immediately. The Scheduling Thread will keep track of the active memory usage, and, in case of memory oversubscription, it will preempt kernels with the most active memory usage. The preempted kernels will not be re-executed until there is enough GPU memory available. To prevent starvation, the system limits the wait time until re-execution to a predefined threshold.

3) Memory Advise (MAdvise): Our Memory Advise scheme mitigates memory-based interference by providing memory hints to the CUDA runtime. In particular, this method leverages the UVM *cudaMemAdvise* primitive, which allows setting a preferred location (i.e., CPU or GPU memory) for a particular range of memory addresses. In MAdvise, the Scheduling Thread keeps track of the active memory usage. Each process is allowed a predefined amount of device memory to utilize. If a process's active memory usage exceeds its provisioned memory limit, then PILOT will invoke *cudaMemAdvise* to hint the CUDA runtime to keep the rest of that process's memory pinned in host memory. This approach could be described as lazy or on-demand eviction. The *cudaMemAdvise* primitive will not evict pages explicitly; a separate event must occur, such as the allocation of new pages from a different process, to evict the pages of the throttled process back into host memory. *A significant advantage of MAdvise is that it does not require kernel preemption*, thus avoiding all associated overheads. When a CUDA kernel finishes running and there is enough device memory to unthrottle another kernel, PILOT will unset the *cudaMemAdvise* hint to allow any pages pinned to host memory to be brought to the GPU device memory.

IV. Experimental Evaluation

A. Experimental Setup

Benchmark applications: We used a set of programs from the Rodinia [7] and PolyBench-GPU [8] benchmark suites and ported them to utilize Unified Virtual Memory. In all cases, we used three input datasets with varying size. The characteristics of the applications and datasets used are summarized in Table II. In order to explicitly control the GPU utilization of each workload, we modified all CUDA kernels to use a configurable number of thread-blocks and a configurable thread-block size.

Memory-intensive Kernel: To evaluate our memory interference mitigation schemes, we wrote a memory-intensive GPU kernel (see Listing 1) which we co-run with the benchmark applications above. Since we are focusing on unified virtual memory with on-demand paging, this kernel sweeps through all the allocated pages to bring them into the device memory and keep them resident there. This execution pattern serves to test the extreme case of memory intensive GPU kernels. This kernel has a configurable memory allocation size and a configurable number of CUDA threads. We encapsulate its body in a forever running while-loop to ensure maximum memory interference. With UVM, the page size is determined by the driver, and the API provides no explicit control over it. Since the smallest page size is 4KB, we define a stride of 4KB conservatively to touch as many pages as possible.

```
__global__ void pageDoS(float *A) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  while(1)
    for( int i=0; i < PAGES_PER_THREAD; i++ )
      A[tid*ELEMENTS_PER_PAGE + TOTAL_THREADS*
            ELEMENTS_PER_PAGE*i] += 1;
}
```

Listing 1: Memory intensive CUDA kernel

System setup: We show the results reported on an NVIDIA Tesla v100 GPU (Volta architecture) equipped with 80 SMs and 32GB HBM. The system used has a Intel Xeon E5 CPU, 128GB memory, and runs Ubuntu 18.04 and CUDA 11.4. We observed similar results on an NVIDIA Pascal GPU.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Benchmark applications and dataset sizes

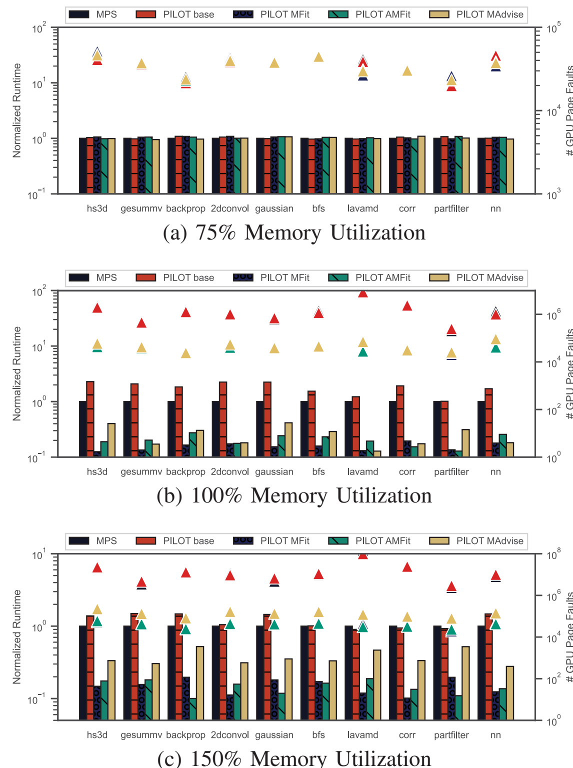
Suite	#	Name	Description	SM Util.	Kern. Exec.	Prob. Size (sm/md/lg)	Mem. Req. (sm/md/lg)
Rodinia	1	backprop	Back Propogation	100%	Long running	65k/1M/3.4M	8.8MB/133.5MB/453MB
	2	bfs	Breath-First Search	100%	Iterative	1M/26M/134M	39MB/1GB/4.9GB
	3	hs3d	Hotspot3D	75%	Iterative	512/2048/4096	48MB/768MB/3GB
	4	partfilter	Particle Filter	63%	Iterative	1024/4096/10240	10.2MB/160MB/1GB
	5	lavamd	N-Body	56%	Long running	10/50/100	7.5MB/936MB/3.3GB
	6	nn	k-Nearest Neighbor	100%	Long running	4M/40M/400M	45.8MB/457.8MB/4.5GB
	7	gaussian	Gaussian Elimination	100%	Iterative	4096/8192/12288	128MB/512MB/1.1GB
Polybench	8	corr	Correlation compute	63%	Long running	2048/4096/8192	64.1MB/256.2MB/1GB
	9	gesummv	Scalar, Vector, and Matrix multiply	100%	Long running	4096/8192/16384	256MB/1GB/4GB
	10	2dconvol	2D Convolution	100%	Long running	252/8192/18432	1MB/1GB/5.1GB

B. Results

We perform two sets of experiments on NVIDIA MPS and PILOT. In the first, we co-run each application with our *pageDoS* memory-intensive kernel. In the second, we co-run applications in pairs. In order to study the performance of PILOT under different degrees of device memory subscription, in each set of experiments we fix the dataset size of the applications (large datasets in Table II) and block off a varying amount of GPU memory (using *cudaMalloc* and *cudaMemset*). We note that, since the GPU reserves some device memory for internal use, 100% subscription is actually a slight oversubscription scenario.

Memory intensive experiments: Fig. 4 shows how PILOT performs when each benchmark application is co-run with the memory-intensive kernel, both without memory oversubscription mitigation (*PILOT base*) and with the proposed mitigation schemes (*PILOT MFit*, *PILOT AMFit*, and *PILOT MAdvise*). We plot, as a bar chart, the overall kernel execution time of each benchmark on PILOT normalized to the execution time on NVIDIA MPS. We also plot the total incurred GPU-sided page faults as triangles on the secondary y-axis. In Fig. 4a, we set the combined memory footprint of the memory-intensive kernel and benchmark application to be around 75% of device memory capacity. We show the memory oversubscription cases in Fig. 4b and 4c (100% and 150% oversubscription, respectively). Oversubscription leads to a substantial increase in execution time and page faults, so for visibility we plot each figure on a log scale for both y-axis. We make the following observations. First, when GPU memory is under-subscribed, MPS, PILOT baseline (i.e., no oversubscription mitigation) and PILOT with mitigation enabled perform very similarly. Second, under PILOT baseline, even slight oversubscription causes a dramatic increase in kernel execution time for all benchmarks. However, through the proposed memory oversubscription mitigation schemes, PILOT is capable of limiting, and in some cases avoiding, the performance degradation, significantly outperforming Nvidia MPS. All three mitigation schemes allow a decrease in page faults.

Multi-tenancy experiments: To evaluate our mitigation schemes in practical scenarios, we co-run pairs of benchmark applications from Table II. We recall that MAdvise does not require kernel preemption, and thus it does not incur any overheads associated with software kernel preemption, checkpoint and recovery support. We measure performance of the following pairs: (i) *corr+backprop*, (ii) *hs3d+2dconvol*, and (iii) *bfs+kNN*. Fig. 5 shows the slowdown experienced by each kernel when co-running over standalone execution (i.e., without inter-kernel concurrency) without and with oversubscription mitigation (through MAdvise). We make the following observations. First, under 75% and 100% memory subscription, all kernels experience little-to-no slowdown in execution time, despite running concurrently. Second, oversubscription causes a more pronounced slowdown, especially for *hs3d* when co-run with *2dconvol* and *bfs* when co-run with *kNN*. In this case, MAdvise is effective in reducing memory contention and, as a consequence, limiting the slowdown experienced. The worse performance of *hs3d* and *bfs* is linked to its irregular memory



(a) 75% Memory Utilization

(b) 100% Memory Utilization

(c) 150% Memory Utilization

Fig. 4: Each benchmark app. is co-run with the memory-intensive *pageDoS* kernel. On the left y-axis we plot execution time normalized to running with NVIDIA MPS. On the right y-axis, we plot the total device-caused page faults.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.

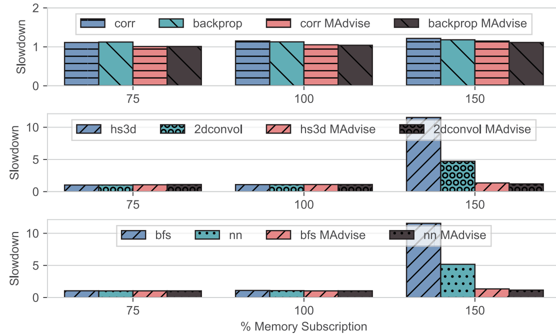


Fig. 5: Multi-tenancy experiments where benchmark applications are co-run in pairs: (i) *corr+backprop*, (ii) *bfs+2dconvol*, (iii) *bfs+kNN*. The charts plot the slowdown experienced by each kernel over standalone execution, without and with memory interference mitigation (MAdvise policy).

access patterns (it has been shown that irregular applications are more affected by memory oversubscription [5]). Not included in this paper due to space limitation, MFit and AMFit are even more effective than MAdvise in reducing contention; however, they incur some additional overhead affecting the total application time due to preemption and rescheduling.

V. Related Work

NVIDIA vGPUs: While offering some performance guarantees, NVIDIA vGPUs incur virtualization overhead and offer limited features do not support all the features without pass-through enabled, in which the entire GPU will be allocated for a single user or context. Currently, missing features of vGPUs include Unified Virtual Memory support, profiling, cuda-gdb, and GPUDirect remote memory access [16].

GPU Runtime Systems: Previous works have proposed runtime systems to support GPU virtualization and multitenancy [10]–[12], [17]–[19], in some cases offering support for concurrent kernel execution [10], [11]. However, none of these systems support UVM and provides memory interference mitigation schemes.

Architectural enhancements: Recent research efforts have proposed architectural enhancements to improve UVM performance and alleviate inefficiencies due to memory oversubscription [3]–[6], as well as hardware preemption support for GPU. [20], [21]. Due to their nature, these proposals have been evaluated using cycle-accurate architectural simulators, and they are complementary to this work.

Runtime optimization for UVM: Li and Chapman [22] proposed a set of compiler-assisted runtime strategies to improve performance of OpenMP applications performing GPU offloading with UVM. Differently from PILOT, their focus is on applications run in isolation. Thus, the proposed runtime does not handle application concurrency and multiple address spaces, and focuses on data placement for a single application.

VI. Conclusion

In this work, we have shown how concurrent execution of applications on GPU can lead to memory oversubscription, which can significantly affect the applications' execution time. We have proposed and implemented three policies that can transparently mitigate performance degradation due to oversubscription of on-device GPU memory. We have developed PILOT, a runtime system similar to NVIDIA MPS that is able to transparently serve as a middleware CUDA runtime. PILOT supports concurrent multi-kernel execution, UVM, active device memory monitoring, and kernel preemption.

VII. Acknowledgement

This work was supported through NSF awards CNS-1812727 and CCF-1741683.

References

[1] R. Landaverde *et al.*, "An investigation of Unified Memory Access performance in CUDA," in *Proc. of HPEC 2014*.
[2] N. Sakharnykh, *Maximizing Unified Memory Performance in CUDA*, 2017. [Online]. Available: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.
[3] C. Li *et al.*, "A framework for memory oversubscription management in graphics processing units," in *Proc. of ASPLOS 2019*.
[4] D. Ganguly *et al.*, "Interplay between hardware prefetcher and page eviction policy in CPU-GPU Unified Virtual Memory," in *Proc. of ISCA 2019*.
[5] D. Ganguly *et al.*, "Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription," in *Proc. of IPDPS 2020*.
[6] H. Kim *et al.*, "Batch-aware unified memory management in GPUs for irregular workloads," in *Proc. of ASPLOS 2020*.
[7] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of IISWC 2009*.
[8] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to gpu codes," in *Proc. of InPar 2012*.
[9] NVIDIA, *Multi-Process Service*, 2019. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA%5C_Multi%5C_Process%5C_Service%20%5C_Overview.pdf.
[10] V. T. Ravi *et al.*, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proc. of HPDC 2011*.
[11] D. Sengupta *et al.*, "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *Proc. of SC 2014*.
[12] M. Becchi *et al.*, "A virtual memory based runtime to support multi-tenancy in clusters with GPUs," in *Proc. of HPDC 2012*.
[13] M. Harris, *GPU Pro Tip: CUDA 7 streams simplify concurrency*. [Online]. Available: https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.
[14] R. Garg *et al.*, "CRUM: Checkpoint-restart support for CUDA's Unified Memory," in *Proc. of CLUSTER 2018*.
[15] O. Villa *et al.*, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. of MICRO 2019*, 2019.
[16] NVIDIA, *NVIDIA Virtual GPU Software Documentation v10.0 through 10.2*, 2020. [Online]. Available: https://docs.nvidia.com/grid/latest/index.html.
[17] V. Gupta *et al.*, "GViM: GPU-accelerated virtual machines," in *Proc. of HPCVirt 2009*.
[18] J. Duato *et al.*, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. of HPCS 2010*.
[19] L. Shi *et al.*, "vCUDA: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2011.
[20] J. J. K. Park *et al.*, "Chimera: Collaborative preemption for multi-tasking on a shared GPU," in *Proc. of ASPLOS 2015*.
[21] Z. Lin *et al.*, "Enabling efficient preemption for simt architectures with lightweight context switching," in *Proc. of SC 2016*.
[22] L. Li *et al.*, "Compiler assisted hybrid implicit and explicit gpu memory management under unified address space," in *Proc. of SC 2019*.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on February 18,2022 at 14:11:08 UTC from IEEE Xplore. Restrictions apply.