ABSTRACT

MANKAD, KARTIK. Analysis of Unified Memory Performance and Protection for Concurrent Kernel Execution. (Under the direction of Dr. Michela Becchi and Dr. Huiyang Zhou).

GPUs are increasingly gaining popularity as a mainstream computing platform due to their excellent performance numbers for modern computing problems, especially for specific application domains like deep learning and virtual reality. They are now commonplace in compute infrastructure offerings from most major cloud computing vendors. It would serve these cloud computing providers well to allow concurrent execution of applications on the same GPU device and thus accommodate more users. However, this is currently not feasible because runtime systems that allow this type of concurrent execution suffer from a lack of memory protection and performance guarantee for kernels executing concurrently. In this work, we study both these shortcomings in the context of the NVIDIA Pascal architecture and the Unified Memory features it offers. We then demonstrate the how a runtime system could be compromised by a Denial-Of-Service attack mounted based on the demand paging functionality part of Unified Memory. As a partial fix, we propose an enhancement to the default memory allocator which is able to prevent the Denial-Of-Service attack. We also discuss possible solutions that could be implemented at the driver level.

Analysis of Unified Memory Performance and Protection for Concurrent Kernel Execution

by
Kartik Mankad

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2018

APPROVED BY:

_____	_____
Dr. Michela Becchi Committee Chair	Dr. Huiyang Zhou
_____	
Dr. James Tuck	

DEDICATION

To my parents, Mehul and Ushma Mankad, and to my beloved fiancée Japa.

BIOGRAPHY

Kartik Mankad was born in Vadodara, Gujarat, India on August 8, 1991. He grew up there and finished high school at Delhi Public School, Vadodara. After high school, he went on to complete a dual degree – a Bachelors in Electrical & Electronics engineering along with an MS in Chemistry at Birla Institute of Technology and Science, Pilani with honors. He joined NVIDIA right out of college as a Verification engineer and worked at their Bangalore Design Center for almost 3 years where he contributed to multiple successful chip tapeouts. He chose to join North Carolina State University in the fall of 2016 to pursue an MS in Computer Engineering. During his time at NC State, he focused his course work on computer architecture and his research efforts on parallel computing systems. He plans to re-join NVIDIA in Santa Clara, California in June, 2018 after completing his thesis at NC State University.

ACKNOWLEDGMENTS

First off, I would like to extend sincere thanks and express utmost gratitude towards my advisor, Dr. Becchi. My research journey as detailed in these pages has been an intellectually stimulating one throughout which I enjoyed great support, candid feedback and generosity from her.

In addition, I would also like to thank Dr. Zhou, whose guidance and encouragement was pivotal in bringing my work to its current state. I can confidently say that his unique insight and in GPU architecture not only helped the end-result but also helped shape the approach leading up to it.

I would also like to thank Dr. Tuck, for agreeing to be a part of my committee and for reviewing this thesis. I am sure that my learnings from course on code optimization are going to be with me for a long time and serve me well in my professional career – for which I am extremely grateful.

Further, I would like to thank my fellow lab mates of the Networking and Parallel Systems lab. I shall always cherish the technical as well as non-technical discussions with them.

Finally, I would like to thank my friends and family for helping me along this journey and helping me back up in my weaker moments. I am fortunate to have them in my life.

TABLE OF CONTENTS

LIST OF TABLES....vi
LIST OF FIGURES....vii
CHAPTER 1 : Introduction 1
Contributions 3
Thesis organization 3
CHAPTER 2 : Background on Unified Memory in Kepler/Maxwell and Pascal Architecture 5
Pre-Pascal Unified Virtual Memory 9
Post-Pascal Unified Virtual Memory 11
CHAPTER 3: Background on Stream-based Concurrent Kernel Runtime Systems 16
The Many Core Runtime 21
The NVIDIA Multi Process Service Runtime 23
CHAPTER 4 : Memory Protection Experiments 25
Methodology 28
Hardware and Software setup 29
Results 30
Conclusion 33
CHAPTER 5: Memory Performance experiments 34
Methodology 37
Results 40
Motivation for an improved allocator 46
Pre-allocator Design 47
Conclusion 49
CHAPTER 6: Future Work 50
CHAPTER 7: Related Work 53
CHAPTER 8 : Conclusion 55
REFERENCES 56
APPENDIX 62
APPENDIX A: Malicious Kernel source code 63

LIST OF TABLES

Table 5.1 Description of the LoneStarGPU benchmark applications used 40

LIST OF FIGURES

Figure 2.1 Flow of execution for a typical CUDA Program 5

Figure 2.2 cudaMallocManaged() in CUDA 6.0 7

Figure 2.3 CUDA program with UVM 8

Figure 3.1 The NVIDIA Driver-Runtime stack 18

Figure 3.2 NVIDIA Driver-Runtime Stack with multiple contexts 20

Figure 3.3 NVIDIA Hyper-Q introduces multiple work queues. 21

Figure 3.4 Many Core Runtime from the NPS Lab 23

Figure 3.5 NVIDIA Multi Process Service 24

Figure 4.1 NVIDIA Driver-Runtime Stack 27

Figure 4.2 Memory Protection Microbenchmark (Serial execution) 28

Figure 4.3 Memory Protection Microbenchmark (Parallel mode) 29

Figure 5.1 Graph showing the slowdown of the victim kernel under the effect of the first version of the malicious kernel 42

Figure 5.2 Bar chart showing the severity of the victim slowdown as malicious kernel fragment size is varied 44

Figure 5.3 Bar chart showing the proportion of time spent servicing page faults across allocation sizes 45

Figure 5.4 Bar chart showing the relative execution speed when LoneStarGPU is run within the proof-of-concept system and MPS Runtime Systems 46

Figure 5.5 Bar chart showing the relative performance of the LoneStarGPU suite when co-run with an application (bfs) from the same suite in the POC system 47

Figure 5.6 Chart showing the effects of a milder version of the malicious kernel on the LoneStarGPU suite as run in the POC system 48

Figure 5.7 Bar chart showing the relative execution time of the LoneStarGPU suite when affected by the malicious kernel but using the pre-allocator 50

CHAPTER 1 : Introduction

Since the introduction of NVIDIA's Compute Unified Device Architecture (or CUDA as it is popularly known) in 2007 [1], its popularity and adoption as a mainstream computing paradigm in industry and academia has seen a sustained increase. Both the programming experience as well as the performance of this system as a whole has improved significantly moving from generation to generation. Apart from the improvements in the programming model itself, this growth of the CUDA computing ecosystem has been supported by its robust and well-designed toolchain that not only assists debugging, but also profiling and deploying code.

Popular IaaS ("Infrastructure as a Service") providers noticed this trend of growing popularity and demand for this platform among their clientele and invested heavily in an attempt to get ahead of the demand curve for this computing platform. It was as early as 2010 [2] just 3 years after CUDA was first launched by NVIDIA, that Amazon Web Services ("AWS") introduced GPU-enabled cloud instances for their popular Elastic Compute service offering.

The mind-boggling amounts of data that need to be processed on a near real-time basis in many commercial and enterprise applications (often dubbed "Big Data") have only underlined the relevance of a massively parallel computing platform like CUDA. Its recent adoption as the de-facto computing platform for deep learning applications is a clear testament to its current and future relevance in the modern computing landscape.

While GPUs represent a financially viable solution to the continuously growing computing needs of tomorrow, it should be noted that the immense parallel processing potential of this hardware platform is shrouded behind the software stack - specifically the runtime layer and the hardware driver itself.

In systems like Microsoft Azure, their GPU-enabled virtual machine instances are setup such that only one user session is allowed to use the underlying GPU card at a given point in time. This mode of managing devices in virtualized systems is referred to Discrete Device Assignment [3]. While the CPU and other attached computing resources on the node(s) might be virtualized across different user sessions, the GPU is attached to one user session only. While this might be intuitive for their users - and convenient to manage from Microsoft's perspective, it is often the case that a lot of the massive computing power available on the node goes unused because that single user might not be running enough compute applications to saturate the GPU's resources. Existing studies [4] have found that this indeed the case many times. They conclude that typical applications (represented by benchmark suites in the referenced papers) often underutilize the available hardware.

A natural next step based on such a conclusion is to try and accommodate enough work on the GPU instances by allowing more than just one user access to the GPU. Not only does this mean that the hardware is better utilized, it also means that commercial cloud computing providers can sell many more times the service hours for the same investment in hardware and computing infrastructure.

The underlying software infrastructure supporting users on such cloud computing systems would need significant changes to accommodate this new use model. Many such systems have been proposed [5] [6] [7] as valiant attempts to fill the void of a robust, multi-user system that virtualizes GPUs at scale to achieve high device utilization. However, all of them suffer from two main flaws that render them unfit commercial deployment - the lack of any performance guarantee and flimsy memory protection for users (if any at all). These are two

basic flaws that must be addressed for any such runtime to make it suitable for use in a production environment.

Contributions

With this brief context of the issues in current runtime systems, the contributions of this work are listed below:

- A study of the extent of the lack of memory protection and absence of any performance guarantee in a concurrent execution scenario on a Kepler and Pascal GPU, both with and without Unified Memory.
- Based on the observations from the above-mentioned study, we develop a malicious CUDA kernel that can create a Denial-Of-Service (DoS) attack on a shared GPU controlled by a stream-based concurrent kernel runtime system. This is benchmarked on the LoneStarGPU [22] benchmark suite.
- We also propose and prototype a runtime system enhancement that represents a moderately successful attempt at limiting the adverse effects of a malicious kernel like the one we created. This too is benchmarked using the LoneStarGPU suite.
- Finally, we lay out the blueprint for what a comprehensive solution encompassing the driver and runtime system to address these issues could look like.

Thesis organization

This thesis is organized as follows: Chapter 2 contains an in-depth description of the Unified Memory feature available on NVIDIA GPUs. Chapter 3 serves to inform the reader about two popular runtime systems that allow concurrent kernel execution. Chapter 4 is about

experiments conducted to evaluate the level of memory protection in the concurrent execution case, with and without Unified Memory. Chapter 5 describes our experiments and microbenchmarks to investigate the performance effects of demand paging. Chapter 6 discusses the future work possible on the basis of the experiments we already describe in previous chapters. Chapter 7 provides a succinct summary of the related publications in this area of GPU research. Chapter 8 concludes.

CHAPTER 2 : Background on Unified Memory in Kepler/Maxwell and Pascal Architecture

This chapter serves to inform the reader about the workings and features of NVIDIA's unified virtual memory, a feature introduced in CUDA 6.0 [8].

In CUDA platforms before CUDA 6.0, the typical flow of a program was as shown below:

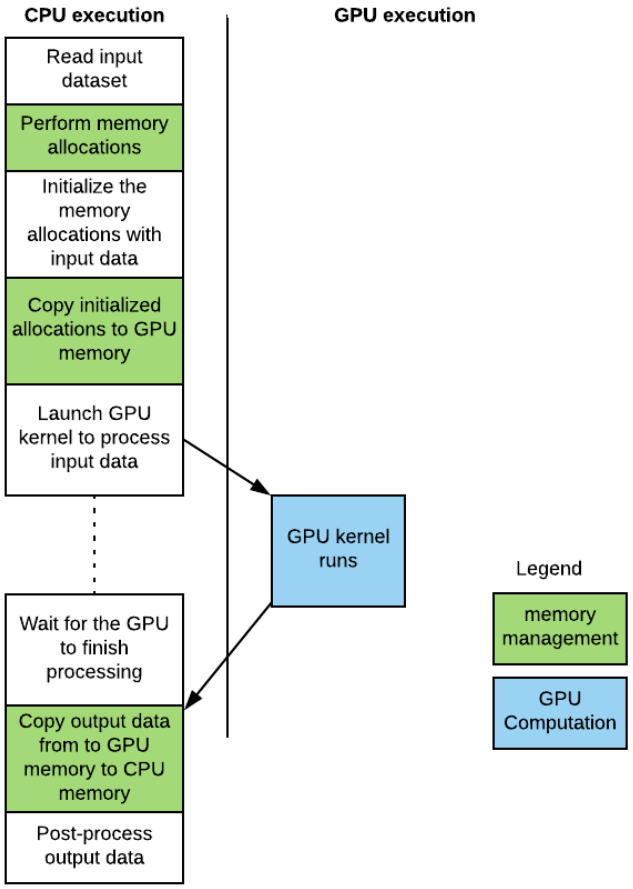

Figure 2.1: Flow of execution for a typical CUDA Program.

The user is expected to setup the working memory for subsequent kernels to process and once done, explicitly program API calls to copy the output data. This layout of most GPU programs is quite intuitive to novice users of the CUDA platform. It also does well to convey the

underlying hardware organization: the GPU and CPU are two, distinct processors and maintain distinct memories. However, it is not the most ideal when it comes to ease-of-use and performance because of the following reasons:

1. Need to setup the copy-in and copy-out operations just right: Mismatch in the number of bytes or pointers often causes hard to track bugs. For simple, continuous buffers this is not particularly hard to get right, but it can get tedious for nested C++ class objects and `structs`.
2. Computational cores idle during copies: While it is possible to use the GPU's copy engines (DMA) to asynchronously copy data into and out of buffers such that it overlaps with computation, setting that up has its own restrictions and is tedious to do in practice. Thus, the scenario where computational cores are idle while data is being copied to/from the GPU is not uncommon.
3. If a program's data set exceeds the physical memory capacity of the GPU, it simply can't be run on the GPU 'as-is'. This prevented a large class of problems across data analytics, genomics and deep learning from being readily deployable on GPUs. The data flow needed to be double or triple buffered for optimal performance. This was until Unified Virtual Memory was introduced.

Considering these inconveniences associated with the traditional copy-compute-copy model, NVIDIA added Unified Memory with CUDA 6. Unified Virtual Memory ("UVM"), as the name suggests, aims to present a unified, virtual address space (and memory management interface) to programmers. Irrespective of what memory (GPU or CPU) a pointer could be referring to, the address space and functions to control the allocation it points to would be the same. Instead of using `malloc()` for CPU memory and `cudaMalloc()` for GPU memory,

users now had the option to use NVIDIA's Unified Memory API - `cudaMallocManaged()`. It could be used for both CPU and GPU allocations. In other words, GPU memory allocated via a `cudaMallocManaged()` call can be seamlessly accessed via pointers even on the CPU. The key is that the system *automatically migrates* data allocated in Unified Memory between host (CPU) and device (GPU) so that the data is seamlessly resident on the CPU memory for code running on the CPU, and on the GPU memory to code running on the GPU. Programmers would never need to call `cudaMemcpy()` in their programs. Thus, the notion of disjoint address spaces and independent memories between the CPU and the GPU no longer exists in the programmer's view with UVM.

```
__host__ cudaError_t cudaMallocManaged ( void** devPtr,
size_t size, unsigned int  flags)
```

Figure 2.2: cudaMallocManaged() in CUDA 6.0.

With UVM, Figure 2.3 shows what a typical CUDA program would now look like. Note that there is no longer explicit copy-to and copy-from operations in Figure 2.3 below.

Further description of UVM is split into two sections based on GPU architecture generations - pre-Pascal (Kepler/Maxwell architecture) and post-Pascal (Pascal/Volta architecture), as Pascal and Volta generations carry significant changes in the mechanisms around UVM.

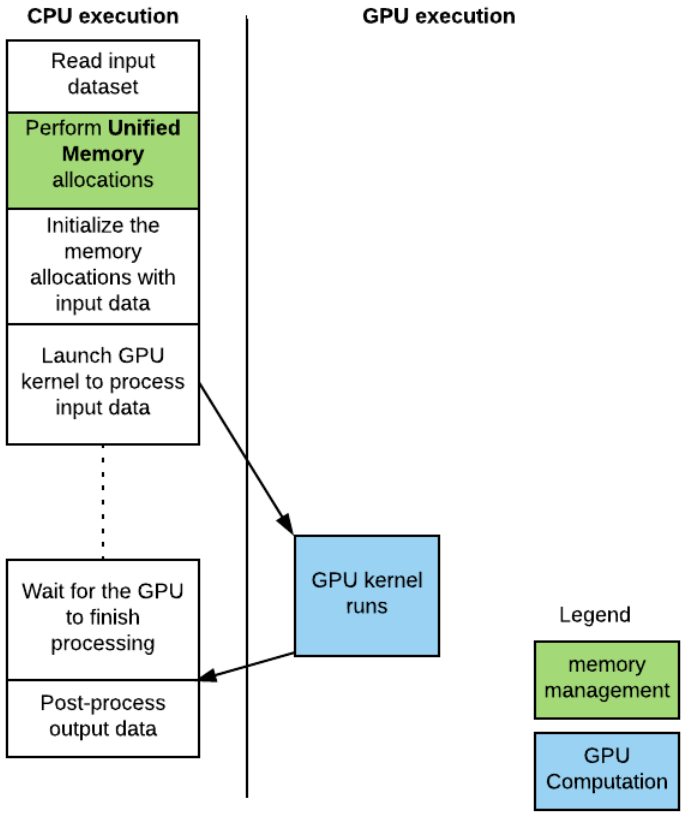

Figure 2.3: CUDA program with UVM.

Pre-Pascal Unified Virtual Memory

In GPU generations before Pascal – Kepler and Maxwell architectures, the Unified Virtual Memory served as little more than an automation layer to the explicit `cudaMemcpy()` calls that the programmer had to make otherwise.

On calling `cudaMallocManaged()` in order to allocate a chunk of Unified Memory, the memory would be allocated on the GPU first. The NVIDIA kernel driver would also setup the page directory and page tables for the allocation on the CPU side, so that any access to this memory would not trigger a segmentation fault, but instead trigger a page fault on the CPU. Initializing these pages on the CPU before launching the GPU kernels (a typical programming practice in many GPU programs) would have those Unified Memory pages resident on the CPU due to CPU page faults that would be triggered when attempting to initialize that Unified Memory allocation.

Further, all allocations made via `cudaMallocManaged()` would be copied over to the GPU by the runtime before the it actually launched the user's kernel on the GPU. Irrespective of whether those allocations would actually be accessed or required by the user's kernel running on the GPU, the runtime system undertook a time-consuming copy operation in order to make all the data available to any code that would run on the GPU. The kernel launch was thus delayed until this copy operation was complete. This was a necessary step for ensuring correct functionality of GPU kernels, but clearly inefficient.

All memory accesses to Unified Memory allocations on the GPU would proceed as before since the runtime system ensures all pages are resident there as before (identical to the

scenario when not using Unified Memory). During a kernel's execution, any access to that allocation by CPU code[1] would return a segmentation fault. The data is accessible from the CPU only after execution on the device is complete, or the device is idle and can handle servicing the copy operation on required pages as requested by the NVIDIA UVM kernel driver operating along with the CPU's page fault handler. These generations of GPUs (before Pascal) could not handle these page copies asynchronously and on-demand.

Assuming the execution on the device was complete and an access was made to a Unified Memory allocation, those pages would be made available on the CPU's memory. The access would trigger a CPU page fault. The default page handler in the OS would then pass on control to NVIDIA's page fault handler (part of the UVM kernel driver) which would recognize this as an access to an allocation of memory created via a previous `cudaMallocManaged()` call. It would then proceed to copy the relevant pages of GPU memory back to the CPU's memory and update the CPU's page table entries to point to this newly-copied-over data in the CPU's memory. [9]

This process is quite like a typical page fault on the CPU, where pages might be swapped out to secondary storage (usually a disk). Instead of the disk, pages are compulsorily swapped out to GPU memory before a kernel launch and are available for swapping back from GPU memory once the device is idle.

---

[1] CUDA Kernel launches are asynchronous with respect to CPU execution. Control returns immediately after the launch operation is complete and does not wait on the kernel actually finishing execution.

Thus, to recap:

- On a `cudaMallocManaged()` memory allocation request : Setup Page Directory, Page Table and mark the pages as not present on the CPU. Allocate that memory on the GPU.
- Before a kernel launch: Copy all Unified Memory allocations from the CPU and to the GPU and set them up on the GPU.
- During a kernel launch: No CPU access to the managed allocations is possible. GPU access would proceed as in the non-Unified Memory allocation case.
- After a kernel launch: The CPU's page fault handler and NVIDIA's UVM driver would work together to copy pages over and set them up on the CPU's virtual memory system on a need basis, per-page.

Post-Pascal Unified Virtual Memory

With the Pascal generation of GPUs, while the programmer visible interface has remained largely the same (with a few additions), but the underlying mechanism has become more efficient in several ways:

1. No immediate allocation: Behind a `cudaMallocManaged()` call, no CPU page table entries are setup, neither is the memory carved out from the available memory on the GPU. Unlike prior generations, these are setup only on an actual access. The current policy is first touch [12]. The pages are setup (and initially resident) on the processor (CPU or GPU) that first attempts to access them. Thus, only the required (accessed) page entries occupy resources in the page management infrastructure of the respective processor.

2. No pre-launch copy: With Pascal and onwards, the pages allocated via `cudaMallocManaged()` calls are no longer bulk copied to the GPU before the kernel is launched. Instead, demand paging is implemented. This means that only on an actual access of the page (and a subsequent *GPU page fault*) will the page be copied from the CPU (if it was resident on the CPU, due to say, initialization) to the GPU. The page could also be resident on a peer GPU and sourced from there. With multi-GPU systems becoming commonplace [10], Unified Memory has been designed with scalability across multiple GPUs in mind.

   Not only does this ensure that there is no overhead before a kernel launch, but also that the GPU's memory is occupied by pages that are really required to reside there. Previously, copying all managed pages was a conservative approach that had the undesired effect of cluttering the GPU memory (and page tables) with pages that might never be used on the GPU in addition to padding the kernel launch with an extraneous overhead.

The key takeaway from #2 above is that GPUs can now raise page faults, just like CPUs can. In the event of a GPU page fault, the respective SM (Streaming Multiprocessor) goes into trap mode and stalls the execution of dependent warps in anticipation of the pages being available once the fault is done being serviced. As part of this sequence of events the corresponding TLB[2] is also locked so that the hardware's view of memory remains consistent across the page fault [12].

---

[2] Depending on which Pascal architecture GPU it is, TLBs may be shared between 2 SMs, or each SM might be tied to a single TLB [11]

The NVIDIA UVM (Unified Virtual Memory) kernel driver then addresses this GPU page fault and executes the below listed sequence of events:

1. Allocate new pages on the GPU;
2. Un-map the page table entries corresponding to the old pages on the CPU;
3. Trigger a blocking copy of those pages from the CPU to the GPU;
4. Setup the respective SM's TLB entries for these newly copied pages on the GPU;
5. Free old CPU page table and TLB entries.

Considering this new GPU page fault mechanism, a number of new features are enabled:

1. Concurrent CPU-GPU access: Unlike the mechanism in Maxwell, Kepler (pre-Pascal), the CPU *can* access memory pages that the GPU might be processing currently. There will no longer be a segmentation fault, but instead a CPU page fault would be triggered. That fault would cause the GPU execution on that page to stall, then trigger an eviction of the page from the GPU's memory system and finally a page-copy operation to have the page now in the CPU's virtual memory system (The sequence listed above). Thus, both the CPU and the GPU can raise page faults over absentee pages and work on the data simultaneously. Ofcourse, while this is functionally possible now, it comes with its own overhead due to the operations involved in a page fault (as listed above). The data copy over the PCIe link between the two processors is the most expensive operation in terms of latency. However, with many page faults, the overhead to update and maintain the paging infrastructure also adds up (Steps 1, 2, 4, 5). This scenario of concurrent access also applies identically to a multi-GPU use case. Multiple GPUs can seamlessly work on the same set of data without the functional need to batch, stream or locally buffer the data. However, for optimal performance those practices might still be required.

2. Over-subscription of GPU memory: This demand paging mechanism also allows programs running on the GPUs to over-subscribe GPU memory. This is akin to how CPU processes may use more than the physically present RAM, by using the secondary storage (usually disk) as a swap buffer. In the case of a GPU, that swap buffer is the CPU's system memory. With this feature, programs whose datasets exceed the physical memory of the GPU can be accommodated "as-is", without the need to setup buffering schemes or batch the processing to make the program fit in GPU memory. To accommodate pages that would otherwise be beyond the capacity of the device, existing pages in the GPU's virtual memory system would be evicted to the CPU's system memory. This eviction would be carried out as part of the GPU page fault itself. This is not optimal from a performance standpoint, but that is far better than not being able to run at all. It provides a very low barrier of entry to being able to start to deploying applications with large memory footprints.
3. System-wide Atomics: For multi-GPU systems such as DGX-2 [10] to be useful to programmers, the support for multi-GPU, or system-wide atomics is a must. This eviction/allocation mechanism of pages ensures that can be achieved. A page can be made to reside on only one GPU's physical memory at any given point in the program's execution. Thus, support for system wide atomics is easily enabled as by the GPU page fault mechanism added in Unified Memory in the Pascal architecture.

From the programmer's point of view, the demand paging mechanism is completely transparent. The only knob that the programmer has been provided is in the form of the `cudaMemAdvise()` call. This call allows the user to specify how they typically expect the page to accessed - whether accesses would mostly be read only or not. Depending on this, the

runtime would simply create a copy of the page on the processor (CPU or peer GPU) that raised a page fault, rather than an eviction from the owner processor followed by a copy operation to the destination processor. Further it also presents the programmer the ability to set a “preferred” processor for a page. Presumably, the driver uses this to keep those pages resident on the preferred processor as far as possible.

The internal heuristics and policies implemented in the NVIDIA UVM kernel driver (`nvidia-uvm.ko`) are not publicly documented. However, the kernel driver itself is open source and the source code can be accessed just like other linux kernel source code. Thus, there exists a lot of opportunity to not only better understand the existing page handling policies implemented, but also implement other policies by modifying this MIT-licensed open source driver to improve the current setup. It must however be noted that the remainder of the NVIDIA driver (`nvidia.ko`) - the part that interacts with the hardware via I/O commands is still closed source. These I/O commands are not publicly documented either. It is only the page handling part of the driver (dubbed the ‘UVM kernel driver’) that is open source. Chapter 4 proposes a policy that could be implemented in the driver to help make the system more robust towards potential memory-based DoS attacks described in this work.

CHAPTER 3: Background on Stream-based Concurrent Kernel Runtime Systems

As a preface to the concept of concurrent execution on NVIDIA GPUs, we shall first explore the typical (non-concurrent) mode of execution for CUDA programs and the associated software layers.

The user's program basically consists of host code that runs on the CPU, and "device" code that runs on the GPU. The device code is usually a set of one or more thread parallel functions that are called CUDA kernels. The primary role of the host code is to control and setup execution of the parallel code that will run on the GPU. Setting up its working data, configuring the device and launching the parallel execution on the device is all to be done by the CPU portion of the program's code. In this sense, the CUDA programming model is CPU centric in terms of the overall flow of execution.

These calls in the host part of the program to manage execution on the GPU are CUDA Runtime API calls. The CUDA Runtime is the layer of software that exists between the closed-source kernel driver and the user program. The runtime interacts with this kernel device driver by calling CUDA Driver APIs. Both the Runtime APIs and Driver APIs are well documented for users - thus allowing them to use APIs at whatever level of abstraction might be suitable for their application. The Runtime API presents a high-level, easy-to-use abstraction of the device, while the Driver API allows users fine-grained control over the execution and configuration of the device.

This CUDA Driver API is implemented in the NVIDIA kernel driver itself. The driver is the last software layer that interacts with the device via proprietary commands over the respective interface (usually PCIe, but NVLink is also supported on some platforms [13]) between the CPU and the GPU.

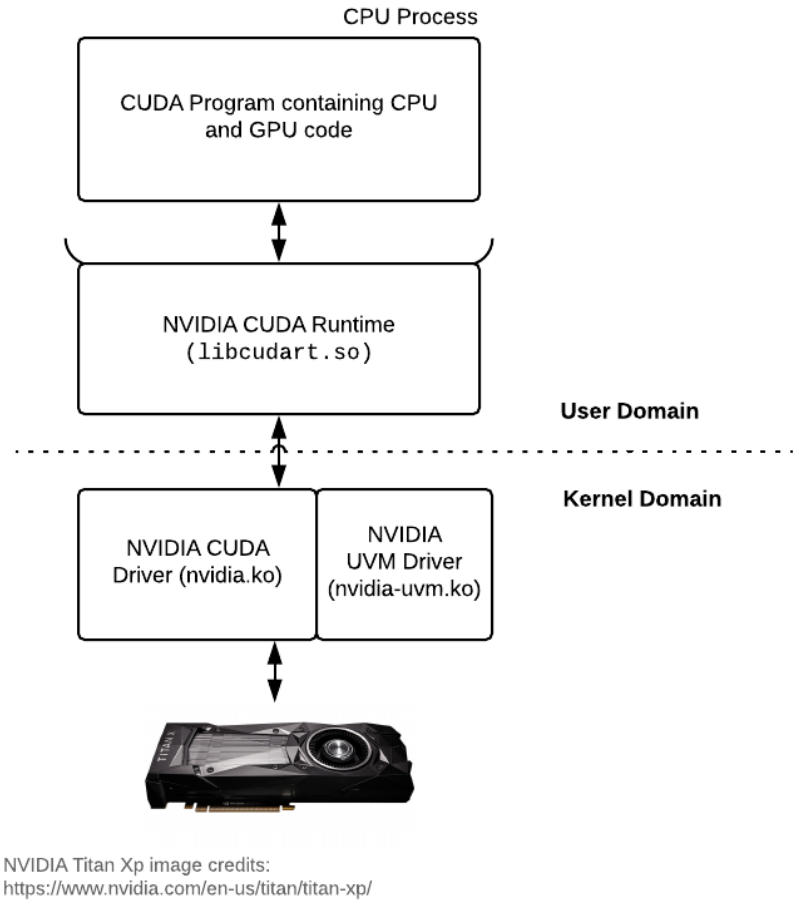

Figure 3.1: The NVIDIA Driver-Runtime stack.

The CUDA Runtime maintains isolation between user processes accessing the GPU by serving those processes within the confines of a software “context”. This context is a per-process software abstraction maintained by the Runtime to allow multiple CPU processes to run device code transparently of each other. Along with independent data structures that represent control

knobs and configuration parameters, each context also presents a distinct virtual address space to the user programs running within its confines.

Since each CPU process is associated with an independent context, there must be a context-switch before the next CPU process can use the device. For devices older than Compute Capability 3.5, the only opportunities the driver uses to Context switch are the end of a kernel's computation, or at the end of a memory copy to/from the GPU. A running kernel cannot be pre-empted. This was a simplistic choice such that very little state information needed to be maintained. For devices with Compute Capability of 3.5-5.*, a running grid may only be preempted at a thread block (or more appropriately in hardware terms, Shared Multiprocessor) granularity. For the device, this means resetting the respective SM pipelines, work queues and associated software data structures in the driver and runtime. This involves some software overhead to keep track of the thread blocks that are pending and running. With devices of Compute Capability 6+ (Pascal, Volta) NVIDIA has introduced Instruction Level Compute Pre-emption. This mechanism allows the execution on the device to be pre-empted at an instruction granularity. This was introduced to help the response time for applications with both compute and graphics requirements (most Virtual Reality applications fall in this category). It involves storing (and restoring) the most amount of state information (register file contents, L1 contents at the very least) owing to this fine granularity compared to previous generations. Irrespective of the granularity, this overhead of "context switching" is prohibitively high to extract satisfactory performance for the case of multiple CPU processes using the GPU running real world compute-only workloads. Analysts have reported it to be over 100x higher the context switch latency on a modern Intel CPU [15]. Running multiple contexts is actively discouraged as far as current best

practices are concerned [14]. Thus, the naive idea of a multi-process CPU program is not a very good way to introduce concurrent execution on the GPU.

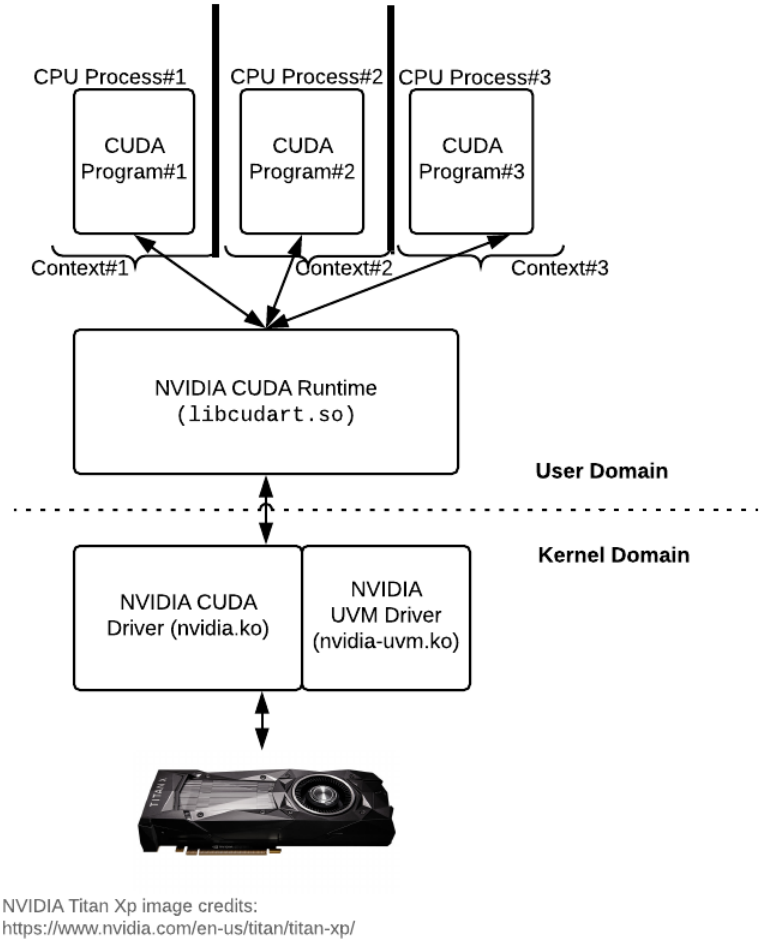

Figure 3.2: NVIDIA Driver-Runtime Stack with multiple contexts.

Early CUDA GPUs supported only thread level parallelism - where users were to write their programs considering what each thread was supposed to execute. While this was an

intuitive model for programmers, NVIDIA added another level of concurrency for programmers to exploit. The Kepler architecture introduced the concept of concurrent CUDA Streams[3] [16]. These Streams represented independent execution sequences that the device consumed. Each such "Stream" in the user's program fed a separate work queue on the hardware. This technology is termed "HyperQ" [16].

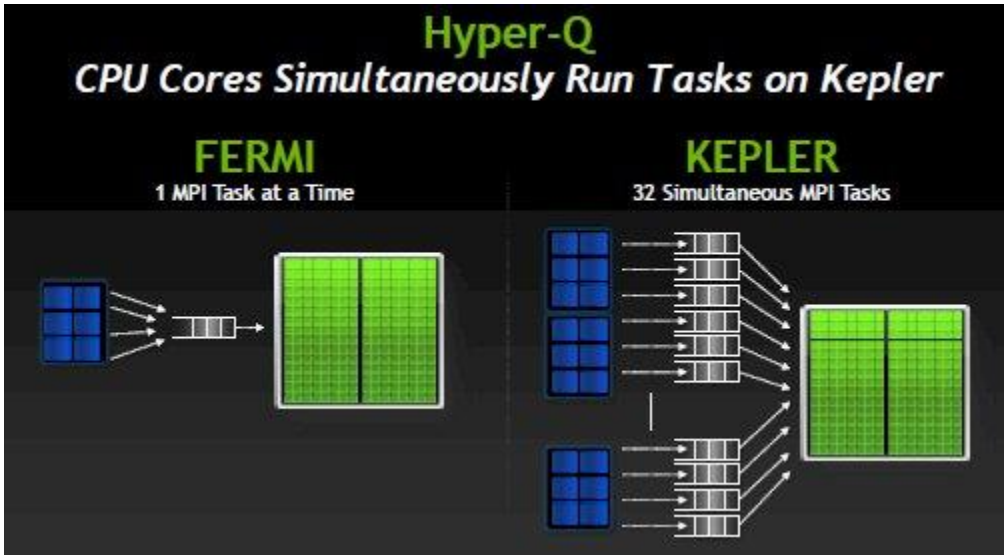

Figure 3.3: NVIDIA Hyper-Q introduces multiple work queues. Image Credits: www.nvidia.com.

This allowed programmers to express asynchronous data transfers and overlap execution from multiple device kernels. One example of an efficient use-model of these Streams was to manage multiple data buffers - one stream to run a copy operation of data from the CPU to the GPU for processing, one stream to execute kernels that would process the data currently on the

[3] While the concept of streams was also present in the Fermi architecture, they were mapped to a single hardware work queue. Thus, they were not truly concurrent. The Kepler architecture introduced *concurrent* streams by having 32 work queues - one per stream which were consumed concurrently.

GPU, and yet another stream to run another copy operation to siphon processed data back to the CPU for post-processing. Such schemes are quite effective in achieving high levels of performance and are only possible because of Streams.

The key takeaway here is that Streams allow for task level concurrency in GPUs. There have been many proposals of runtime systems that leverage this feature to provide the ability to run multiple kernels (presumably from different users/processes) on the GPU concurrently. In the following pages, we shall briefly elaborate on two such systems - the Many Core Runtime from the NPS Lab [17] and the Multi-Process Service developed by NVIDIA [18].

The Many Core Runtime

The Many Core Runtime ("MCR") system proposed by Dr. Becchi et. al [17] is a robust stream-based runtime system. Apart from virtualizing the GPU by allowing multiple processes to share the GPU by leveraging CUDA Streams, the runtime supports the following novel features:

1. Dynamic binding between an application and device: This allows the runtime to spread the execution of kernels from the same program across multiple GPUs - thus achieving near-ideal device utilization.
2. A smart memory management system that allows kernels with memory footprints that might exceed the device capacity to run seamlessly. The runtime performs the necessary data swapping needed to achieve this in the background. This is noteworthy because it was proposed and implemented successfully at a time when the CUDA platform did not support Unified Virtual Memory with all its features as it currently does today.

3. In addition to these, this runtime was also built to support load balancing in case of GPU addition and removal as well as provide some level of resilience to GPU failures.

Since its initial version, this system has seen improvements [19] in terms of optimizing programs' inbuilt synchronizations and thus improving overall device utilization. However, this system suffers from the lack of memory protection and does not carry any mechanism to regulate the execution of multiple kernels such that each of them get a fair share of execution and performance.

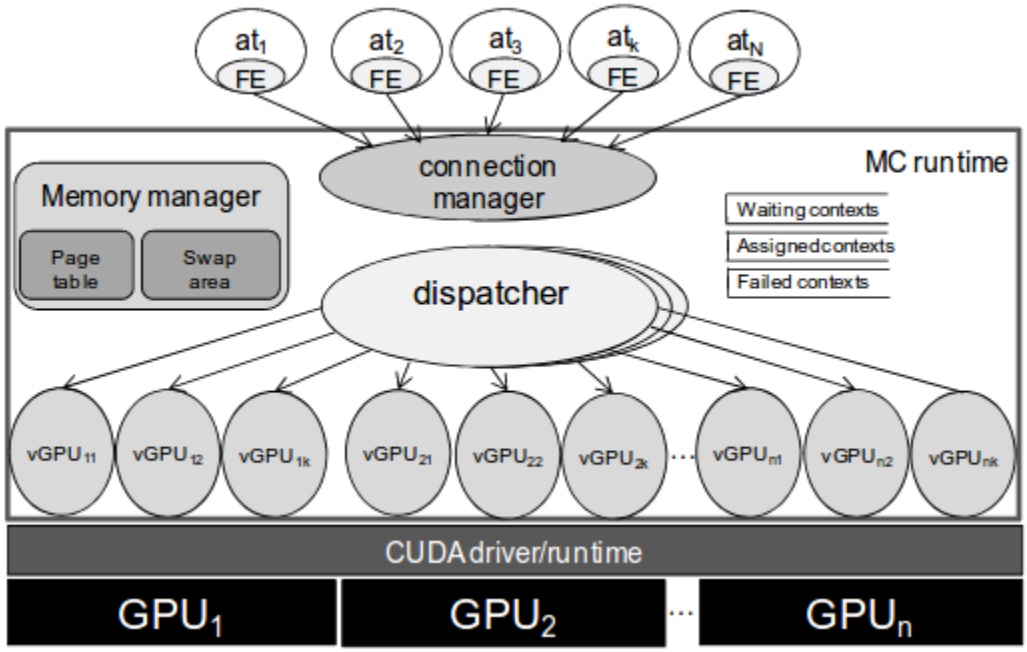

Figure 3.4: Many Core Runtime from the NPS Lab. Image Credits to the original authors of [17].

The NVIDIA Multi Process Service Runtime

The Multi Process Service Runtime ("MPS") developed and maintained by NVIDIA [18] is also a stream-based node-level runtime. It is a Linux-only system that is available on Kepler and newer GPUs. While this runtime system was developed to allow MPI processes to seamlessly use the same GPU device - it serves the purpose of running concurrent kernels irrespective of whether they are part of an MPI process or not.

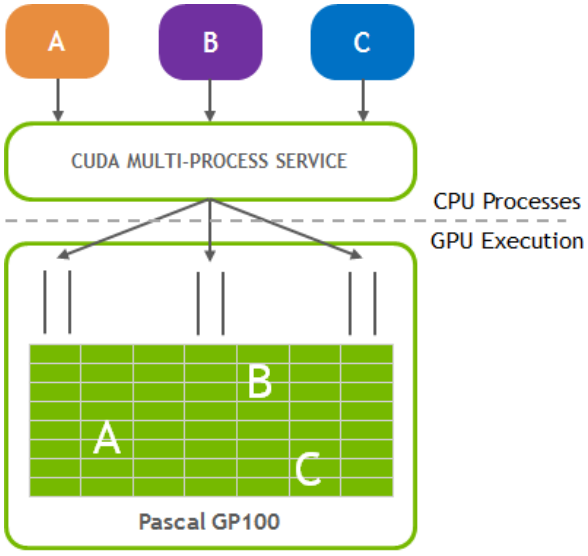

Figure 3.5: NVIDIA Multi Process Service. Image Credits belong to NVIDIA Corporation

The current documentation for MPS states that it is only intended as a runtime for cooperative processes effectively acting as a single application, such as multiple ranks of the same MPI job. Considering this limited scope of deployment, it carries only basic mechanisms to enforce some performance guarantee and memory protection.

For ensuring that one job does not dominate the rest in terms of performance, the MPS runtime allows the user (administrator) to set the maximum number of threads that an MPS Client (associated to a CUDA Stream internally) can launch. If a kernel launches too many

threads, it would most certainly dominate execution and prevent other kernels from even launching on the device due to insufficient resources being available to them. The system however does not enforce any real-time checks, or execution-based mechanisms or policies to ensure the relative performance of running jobs.

On the memory protection front, the MPS runtime for Volta and beyond claims to maintain fully isolated virtual address spaces for concurrently running jobs. This claim has not been investigated in this work since our experimental focus is on the Pascal architecture. For pre-Volta architectures (Pascal, Maxwell, Kepler) the runtime has no mechanism to raise errors for "out-of-bounds" memory accesses by potentially malicious kernels. Accesses made with an intent to corrupt or read the working memory of other kernels running on the device can proceed undetected. Though it is unable to restrict rogue pointer access, it does prevent explicit corruptions attempts made via `cudaMemcpy()` calls to access/modify the working memory of another job which was also setup via a set of one or more prior `cudaMalloc()` and/or `cudaMemcpy()` calls. The experiments on memory performance detailed in Chapter 4 are also carried out on the MPS runtime for completeness sake.

In this work, we create a 'proof-of-concept' runtime system (referred to as POC Runtime in the following pages) setup which is inspired by these two systems for our experiments. It consists of a simple multi-threaded single process application that runs benchmarks within it. Those applications are mapped onto separate concurrent streams, as in the MCR system. However, we do not employ any of the sophisticated scheduling policies the MCR system does.

CHAPTER 4 : Memory Protection Experiments

This chapter describes various experiments and microbenchmarks created to better understand the extent to which the lack of memory protection can be exploited on the Pascal architecture for programs running concurrently on independent CUDA streams on the same device. The general approach followed was to come up with a possible “malicious” kernel for various scenarios and see how it could attack/compromise a potential victim kernel’s execution.

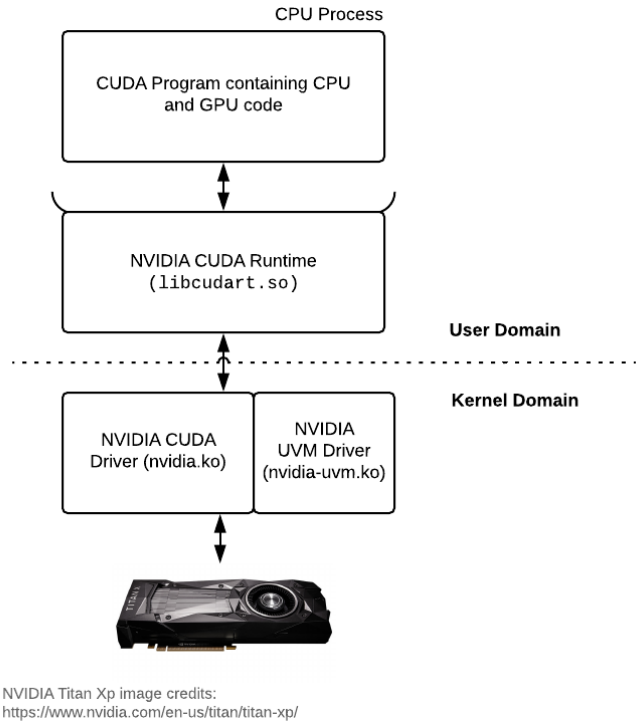

Figure 4.1: NVIDIA Driver-Runtime Stack.

There are two types of exploits that we attempted for memory protection – ‘Serial’, and ‘Parallel’ as described below:

1. Serial : ‘Serial’ here implies that our malicious kernel was after the victim kernel fully completed execution. The GPU was also reset between these two runs using the `nvidia-smi` tool. The figure below shows the execution sequence:

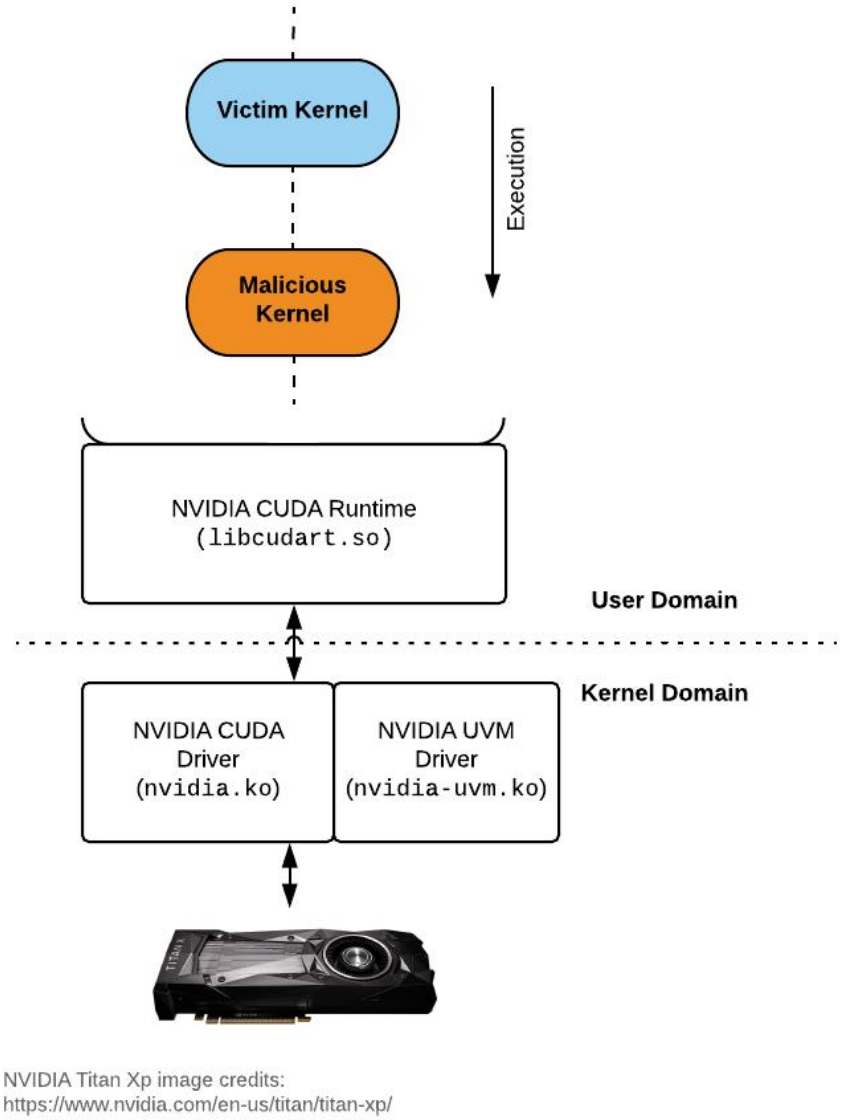

Figure 4.2: Memory Protection Microbenchmark (Serial execution).

The intent of the malicious kernel here is to attempt to read/recover the data “left over” by the previous kernel by performing what is essentially a memory dump of the GPU’s memory. This attack was also repeated with the victim and the malicious kernel running in different contexts (as different CPU processes).

2. Parallel: In this case, our victim kernel ran concurrently with the malicious kernel. The malicious kernel aims to corrupt the working set of the victim in real-time as it is executing. Just like the “Serial” attempt, this too was repeated for the case of different contexts.

The figure below shows the sequence of execution on the device:

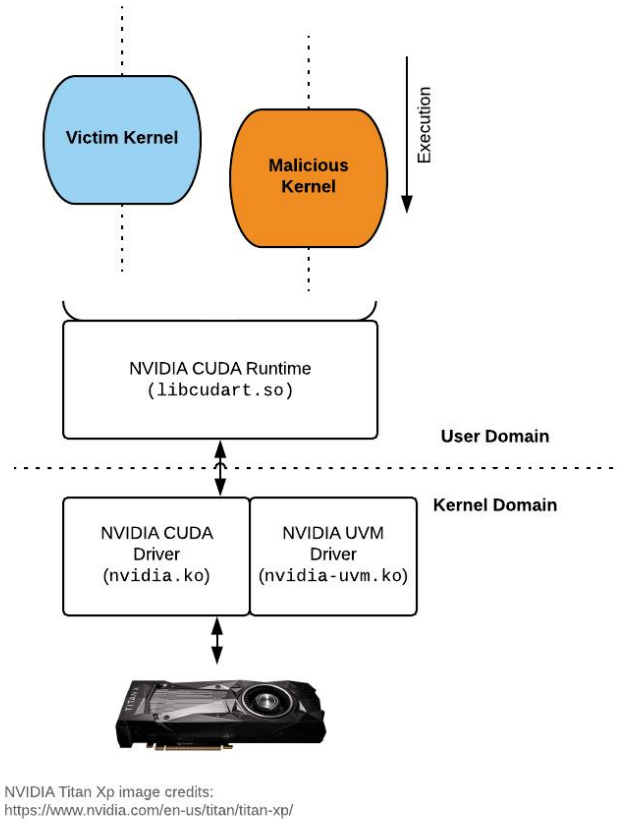

Figure 4.1: Memory Protection Microbenchmark (Parallel mode).

For both these cases, we chose a simple vectorAdd (from the NVIDIA CUDA SDK Sample suite) kernel as the victim kernel. While this might be too simple a kernel compared to real world applications, it is adequate to demonstrate the vulnerabilities we are interested in.

Also, the experiments were repeated for both cases – using Unified Memory (using `cudaMallocManaged()`) and using traditional allocations (`cudaMalloc()`) . Further, we attempted to accommodate the major classes of GPU architecture based on developments in the Unified Memory feature-set – pre-Pascal (Kepler in this case) and Pascal. A Tesla K20c was used as a representative of the pre-Pascal Kepler architecture and a TITAN Xp as a candidate from the Pascal family.

Methodology

For the experiments involving in the same context, we simply compiled both the malicious and victim kernels together into one application and minimally modified their launch calls to run on separate streams. Neither used the default Stream, Stream 0. This was done to maintain the experimental condition of a single context that would exist in a full-fledged runtime system like the Many Core Runtime. Where needed, we introduced minimal modifications to convert the vectorAdd program to use Unified Memory. Allocations were always free'd once they were no longer needed using the `cudaFree()` call.

For the case involving running in separate contexts, we compiled these applications separately and launched both depending on the type (serial/parallel) of exploit. The parallel case was not excessively controlled – apart from the condition that the kernels should indeed run

concurrently, no additional synchronization conditions were imposed to keep the conditions as realistic as possible.

There is no quantitative metric we tracked for these protection tests, but only whether the malicious kernel was able to recover the data (even if partially) of the victim kernel, and whether the malicious kernel was able to corrupt the execution of the victim.

Hardware and Software setup

The experiments described above were run on a node with two NVIDIA GeForce Titan Xp GPUs for when the Pascal architecture was evaluated. Only one GPU was utilized for our runs. Each Titan Xp GPU was equipped with 3840 CUDA cores clocked at 1.6GHz with a total memory bandwidth of 547.7Gbps to 12GB of GDDR5X on-board memory. For experiments involving pre-Pascal GPUs, we used a Tesla K20c. The K20c is equipped with 2496 CUDA cores clocked at 706Mhz with a total memory bandwidth of 208Gbps. Frequency boost was disabled for these experiments. ECC was also kept off for most of the runs. The compute node also had two 6 core Intel® Xeon® E5-2630 CPUs with each core clocked at 2.2GHz. The CPU memory capacity was 125GB. The CUDA SDK version used was v9.1.85 and the GPU kernel driver version number was 390.30.

Results

This sub-section is organized based on the type of exploit attempted (serial or parallel) and the number of contexts involved (same or multiple):

1. Serial:
    a. Same Context, Different Streams: Running a malicious program within the same context after the victim kernel completed allowed us to completely read the memory contents as they were left by the previous kernel when it exited if the malicious kernel requested *exactly the same sized memory allocations and did not use Unified Memory*. The use of unified memory irrespective of the architecture generation (Kepler/Pascal) did not have the same outcome – we were unable to recover the victim kernel's data. This seems to indicate that the traditional memory allocator does not employ any sort of address layout randomization, but the Unified Memory allocator does. Digging into the source code of the NVIDIA UVM driver, we observed that the CPU side page allocations were made using the Linux kernel's `alloc_pages()` API – which does randomize address layout. We were unable to deduce from the GPU side allocation functions' code whether there was intentional randomization, but the results certainly seem to indicate there is. We also observed that the GPU reset via the `nvidia-smi` tool did not have any effect on the memory contents, with all other GPU settings remaining unchanged between the victim's run and the subsequent malicious run. Also, performing `cudaFree()` does not seem to affect the memory contents themselves in the case where data recovery was successful.

b. Different contexts: Here also, we were able to exactly recover the memory contents of the victim kernel as long as the malicious kernel requested exactly the same memory sizes as the victim and did not use Unified Memory. The same physical pages were assigned (owing to the lack of address layout randomization) and simply reading those pages allowed us to recover the data.

c. Another accidental observation made was the initialization of the ECC (Error Correcting Code) functionality for the GPU memory (either done when the feature is turned on, or on what is presumably driver/runtime[4] initialization if the feature is already on.) effectively triggered a zero-ing out of the GPU memory to bring the GPU's memory error tracking logic to a known state. Thus, if ECC is toggled between runs, the data is rendered unrecoverable between runs. Support for memory ECC exists primarily on the datacenter class Tesla product line only, so this was discovered when the same tests were run on a Tesla K20c GPU via distinct CPU processes, serially. In most Tesla GPUs, ECC protection is enabled by default. Thus, when the context was re-created for the subsequent run, it triggered an ECC initialization that reset the state of memory. It is worth pointing out that toggling ECC support requires administrative privileges and access to the `nvidia-smi` tool.

[4] We expect the ECC initialization to be in the driver's domain rather than the runtime – and out of the program's control, but we don't know for sure since there are no published details about its inner workings.

2. Parallel:
    a. Same context, different streams: In this case, the absence of dedicated address space for each kernel allowed the malicious kernel to corrupt the working set of the victim quite easily once the victim's memory layout was known. The malicious kernel could simply zero out the data structures maintained by the victim. However, the knowledge of its layout before-hand is a weak assumption (and a necessary pre-condition) and may not often occur in practice. Knowing the memory pointers of the victim's data structures is a necessary pre-condition for such an exploit because arbitrary memory accesses will be flagged as erroneous accesses by the device and will not succeed – irrespective of the type of allocation made (Unified or traditional). Such illegal memory accesses would abort execution of the malicious kernel at the very least and sometimes abort *all* execution on the device in some cases. That in itself could be viewed as a primitive denial of service attack, but it orthogonal to the current scope of our current experiment - To corrupt the running memory of a victim kernel.
    b. Different contexts: In this case, the presence of separate virtual address spaces completely isolated the victim kernel from the malicious kernel's attempts to influence its execution by memory corruption. In this case, memory protection indeed exists and works as intended irrespective of the type of allocation used. However, the presence of multiple contexts incurs the penalty of context-switching as described earlier in Chapter 3. It should also be interesting to note that even though both kernels ran in different contexts, a large non-

Unified Memory allocation by one kernel could cause the other kernel's similar non-Unified allocation to fail due to inadequate free memory on the device. Allocating these as Unified Memory allocations does however work around this possible issue – another reason to use Unified Memory. This again could be the premise of a simple Denial-Of-Service attack on a concurrent runtime system for applications using the traditional `cudaMalloc()`, but it is an exploit that the runtime system can be easily enhanced to prevent as discussed in the section on the performance experiments in pages ahead.

Conclusion

Thus, our tests on memory protection reveal a drawback of the current traditional memory allocator - specifically the lack of address space randomization. Further, we learn that the runtime does not attempt to clean up the memory after execution is complete and this is revealed by the lack of address randomization in the non-Unified Memory case. We also noted that Unified Memory is superior to the traditional allocator since we were never able to recover data at all when Unified memory was used, irrespective of the architecture involved. The only time that memory is physically cleared is when ECC is first turned on.

Thus, we can conclude that though a potential vulnerability exists, exploiting it in practice needs more work since a raw dump might not readily reveal clues about the organization and significance of specific data structures of the victim kernel.

CHAPTER 5: Memory Performance experiments

Our approach to formulate possible exploits aimed at affecting a victim kernel's performance is divided into two sub-categories as listed below.

1. Compute/resource based
2. Memory based

Let us look at both these categories:

1. Compute/resource based: The negative side-effect that a malicious kernel could have in terms of resources would be to occupy the lion's share – simply with the intent of keeping them unavailable for the victim kernels to use. If the resource requirements for the victim kernel are not met simultaneously, it will not be launched at all. Till the requirements are not met (which could be never – depending on how unrelenting the malicious kernel might be), the victim kernel is kept shelved in the work queue thus completely ruining its turnaround time. These resources that the malicious kernel could claim-and-hold could be a large fraction of shared memory and/or registers, and/or number of threads.

   This form of influence on the victim kernel is easily understood. It may also be controlled easily from a runtime point of view. Both the malicious kernel and the victim kernel must go via the runtime to launch their kernels, setup their data structures etc. The runtime system could then set and enforce limits on the resources that could be allocated to any single kernel to prevent this exact scenario of a malicious kernel grabbing all or enough available resources. The Driver API exposes function calls to query the register and shared memory usage of a GPU kernel - specifically `cuFuncGetAttribute`() [21]. A runtime system could leverage this

and enforce certain limits on kernels. This also includes limiting the launch grid parameters (number of thread blocks and number of threads per block) which the runtime could enforce per-kernel or per-stream limits on. The NVIDIA MPS runtime does claim it implements such a configurable limit on the number of threads a kernel could execute [18]. We have not implemented any such measure in the MCR runtime in this work but recognize it as a future enhancement that can be implemented using the function suggested above.

2. Memory based: Our scope of investigation for this category is quite specific. The introduction of demand paging in the Pascal architecture, or more specifically the ability of the GPU to raise page faults opens the door to a memory page fault based exploit that might be able to severely degrade the performance of the victim kernel. We ran experiments where we created conditions to generate increasing numbers of page faults on the device, thus adversely affecting the victim kernel's performance as it executes concurrently with the malicious kernel.

   The basic idea behind this malicious kernel is to force the victim kernel's execution to slow down by creating a situation in which it (victim) is forced to raise page faults at almost every memory access it makes. Each time-consuming page fault would have the SM's execution pipeline stall till the fault is serviced, thus slowing down the overall execution of the victim kernel.

The way our malicious kernel achieves the above described effect is by doing large, oversubscribed allocation(s) and keeping all of the pages from those allocations alive. Once launched, the malicious kernel would touch each page in this oversubscribed allocation(s) over and over. These incessant accesses to each page would force the UVM driver to keep those pages

alive in the GPU's TLB, page tables and physical memory. Since the allocation is larger than the physical memory capacity of the devices, the virtual memory system would be completely saturated because of these accesses by the malicious kernel. A simple for-loop with page access (read/write) achieves this. For our hardware setup, the memory capacity of the GPU was 12GB. Thus, we are talking about an allocation that totals up to more than 12GB.

To bring in (and keep) the malicious kernel's pages on the GPU in response to its accesses, the UVM driver would be forced to evict pages from the victim kernel's working set - thus forcing the victim kernel to incur a page fault when the victim kernel accesses the now-evicted pages later in the future. It could also be possible that the UVM driver chooses to evict some pages from the malicious kernel's working set. However, this does not affect the outcome of this exploit due to the repetitive nature of the accesses that we plan to have the malicious kernel do.

Another important key point to note here is that - The GPU page size is *not fixed*. It is not always the case that the GPU chooses a page size identical to the CPU. So, it is also not the case that the transfer of the contents of a page in response to a page fault is exactly the CPU page size. The NVIDIA UVM kernel driver appears to employ some heuristics to choose an appropriate page size. Existing literature for the Pascal architecture [24] states the page size is at least the OS page size but could be larger too. It is possible that this sizing is architecture dependent, but we have not explored other architectures' page sizing heuristics in the UVM driver.

Methodology

The hardware/software setup is like the memory protection experiments, with the main distinction being that we ran all the memory performance experiments on the TITAN Xp only since we are interested in the demand paging aspect of the Pascal architecture.

Unlike the memory protection experiments, we are interested in quantitative data in this case, specifically the wall clock time to complete execution. An initial set of experiments was run with vectorAdd as the victim kernel, whose footprint was kept at 3GB. Keeping the kernel simple and well understood for the initial set of experiments was key, since it helped us fine tune the scenario and eliminate secondary effects from contributing to the results. Further, profiling a simple kernel like vectorAdd is much easier and allows us a clear picture of the execution flow – unlike, say a benchmark application that might have multiple kernels, streams, overlapping copies and code that might be creating secondary effects which might unnecessarily influence the results of our initial experiments.

From these initial experiments, we deduced an ‘optimal’ configuration of the malicious kernel. Here, the ‘optimal’ configuration was one that could slow the victim to a glacial pace by creating a near Denial-Of-Service situation for our simple victim. Once this ‘optimal’ formula for the malicious kernel was isolated, we repeated these experiments with real world benchmarks from the LoneStarGPU suite [22] running them via our ‘proof-of-concept’ runtime (POC Runtime) to see if we got similar results.

Before we could run the applications from the LoneStarGPU suite concurrently with our malicious kernel, the applications needed updates to use Unified Memory instead of the traditional allocation & explicit memory copy APIs. The existing Unified Memory port created

by A. Vastrad et. al under the guidance of Dr. Zhou was leveraged with some minor modifications[5] [23]

To carefully isolate the cause of any changes in the measured wall clock time for these applications, we established multiple baselines. First, we obtained the wall clock times when the benchmarks were run standalone – as regular applications with the default NVIDIA runtime. Then, we measured the wall clock time for the applications when run inside the POC Runtime system as well as the NVIDIA MPS runtime system (to get a comparative idea of the overhead involved when running these applications within a non-default runtime system) without any concurrent kernel co-executing with them. We then attempted to get an approximate measure of the slowdown observed when an application is co-run with another, non malicious application. We chose the bfs benchmark in the suite as the concurrent execution companion to the rest of the suite. This way, we can be absolutely sure that the subsequent results are only because of the malicious kernel and can verify that the execution scenarios playing out are exactly in line with our experimental intent here. Finally, we obtained the baseline wall clock time for the applications in the benchmark suite when they were co-run with two representative configurations of the malicious kernel's allocation pattern – one below device memory capacity (in our case, less than 12GB so that no oversubscription occurs) and one above (oversubscription occurs) to see the effects of our worst malicious kernel with real world applications in a concurrent execution scenario.

[5] The initial ported code carried some extraneous cudaDeviceSynchronize() calls in some benchmarks that were possibly added for debugging. These were removed to ensure that the flow of the program remained unchanged with respect to GPU execution.

Each benchmark in the LoneStarGPU suite has multiple flavors because of different input datasets and options. The table below shows the input configurations and kernel dimensions used:

Table 5.1: Description of the LoneStarGPU benchmark applications used

Name	Description	Inputs	Largest Kernel Dimensions
bh	Barnes-Hut Force calculation algorithm	Number of bodies 3000000, TimeSteps 2	Grid:60x1x1 Block:1024x1x1
bfs	Breadth first search using a modified Bellman Ford algorithm	USA Florida Road Network Graph (~1M nodes, ~2.7M edges)	Grid: 104x1x1 Block:1024x1x1
dmr	Delunay Mesh Refinement	Synthetic mesh with 5M nodes	Grid: 420x1x1 Block:64x1x1
pta	Points-To analysis of memory pointers represented as a graph	'Pine' benchmark graph with 6M nodes and 4M edges	Grid: 90x1x1 Block:32x16x1
mst	Minimum spanning tree using Boruvka's algorithm	USA Florida Road Network Graph (~1M nodes, ~2.7M edges)	Grid: 1046x1x1 Block:1024x1x1

Results

The results from our initial experiments are as graphed below: The malicious kernel's continuous memory allocation pattern is varied from 3GB to 12GB. It is concurrently executed with the vectorAdd kernel in a separate but independent stream as described earlier. The launch parameters were kept such that both, the victim as well as the malicious kernels could be accommodated concurrently on the GPU. Both kernels were granted an equal number of thread blocks (15 each, out of the 30 total on the NVIDIA TITAN Xp card) to ensure that they ran concurrently. Their concurrent execution was then confirmed using the NVIDIA Visual Profiler as well.

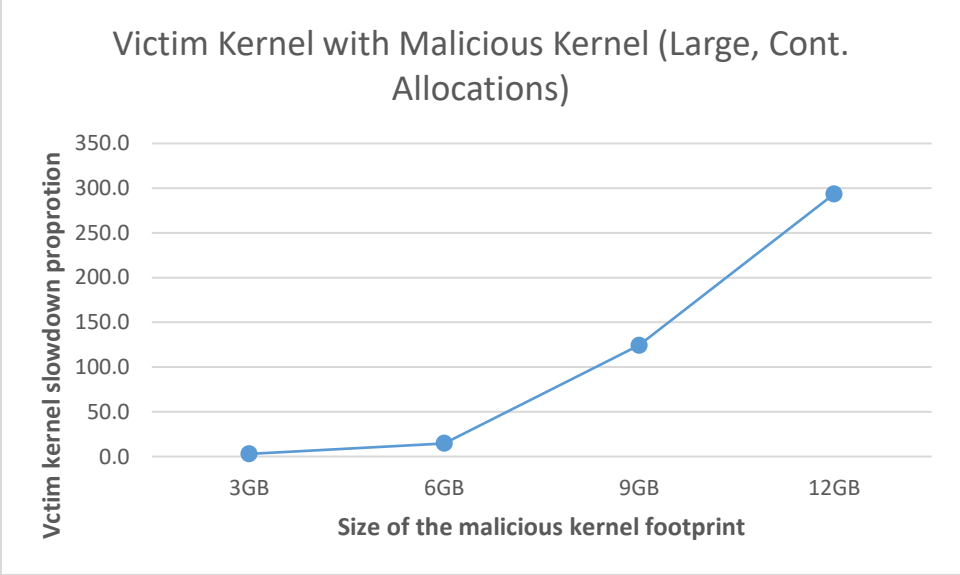

Figure 5.1: Graph showing the slowdown of the victim kernel under the effect of the first version of the malicious kernel.

We see a clear correlation between the allocation size and the slowdown in the vectorAdd kernel's execution time. This is inline with our expectations since larger allocations by the malicious kernel imply a proportionally larger share of the GPU's virtual memory system would

be hogged by the malicious kernel. As allocation sizes increase and the malicious kernel nearly saturates the virtual memory system, the victim kernel faces greater difficulty in executing without experiencing numerous page faults – slowing it down significantly.

Tweaking the malicious kernel [Appendix A], we observed that increasing the trip count of the for loop used to touch pages within the allocation helped prolong the effects of the malicious kernel's dominance on the GPU's memory system. To improve the malicious kernel further, we reasoned that we needed to create a scenario where even greater numbers of page faults were seen. One way we could increase the number of page faults for the same number of accesses (unchanged allocation size and access loop trip count) would be to decrease the page size. Unfortunately, the page size is not something the user can control. The page size is decided by internal heuristics in the NVIDIA UVM kernel driver. However, we did observe that smaller allocations (of the order of KBs) hinted the internal heuristics in the driver to pick the smaller options among its possible choices for the GPU page size. Similarly, larger allocations (of the order of GBs) could nudge the UVM driver into picking larger sized pages to store these larger allocations. Thus, to trick the UVM driver into using small pages, we fragmented the malicious kernel's allocation while keeping the overall footprint large enough to achieve oversubscription. Rather than one large allocation, we decided to allocate multiple smaller arrays, the combined memory footprint of which would be larger than the device's memory capacity – thus maintaining oversubscription. The malicious kernel with these updates slowed down execution of the victim kernel to the extent it gave the user the illusion[6] that the victim was hung – a

[6] We use the word 'illusion' to be conservative. No visible error (dmesg, strace, STDOUT) is seen, and execution doesn't proceed either. We observed that the driver seems to hit an 'unknown fatal error' by examining the logs of

Denial-Of-Service situation. The results of the malicious kernel with fragmented allocations but a shorter trip count are shown below:

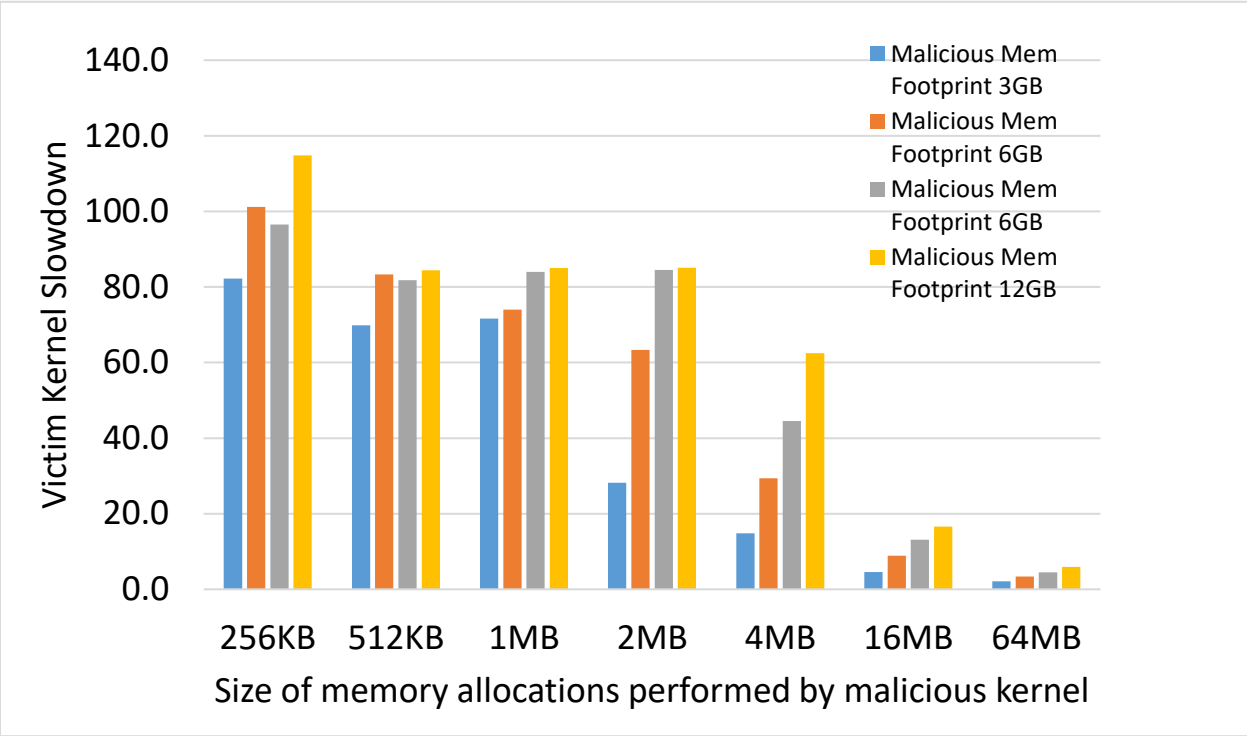

Figure 5.2: Bar chart showing the severity of the victim slowdown as malicious kernel fragment size is varied.

---

the NVIDIA MPS Runtime system's and can't seem to recover. Apart from the fact that no execution seems to proceed after his error, not much else is known. No such error is seen when the malicious kernel or any of the victims are run standalone with MPS or POC runtimes.

The chart below shows the proportion of time spent servicing page faults during the execution relative to the case when the victim ran alone, without a malicious kernel.

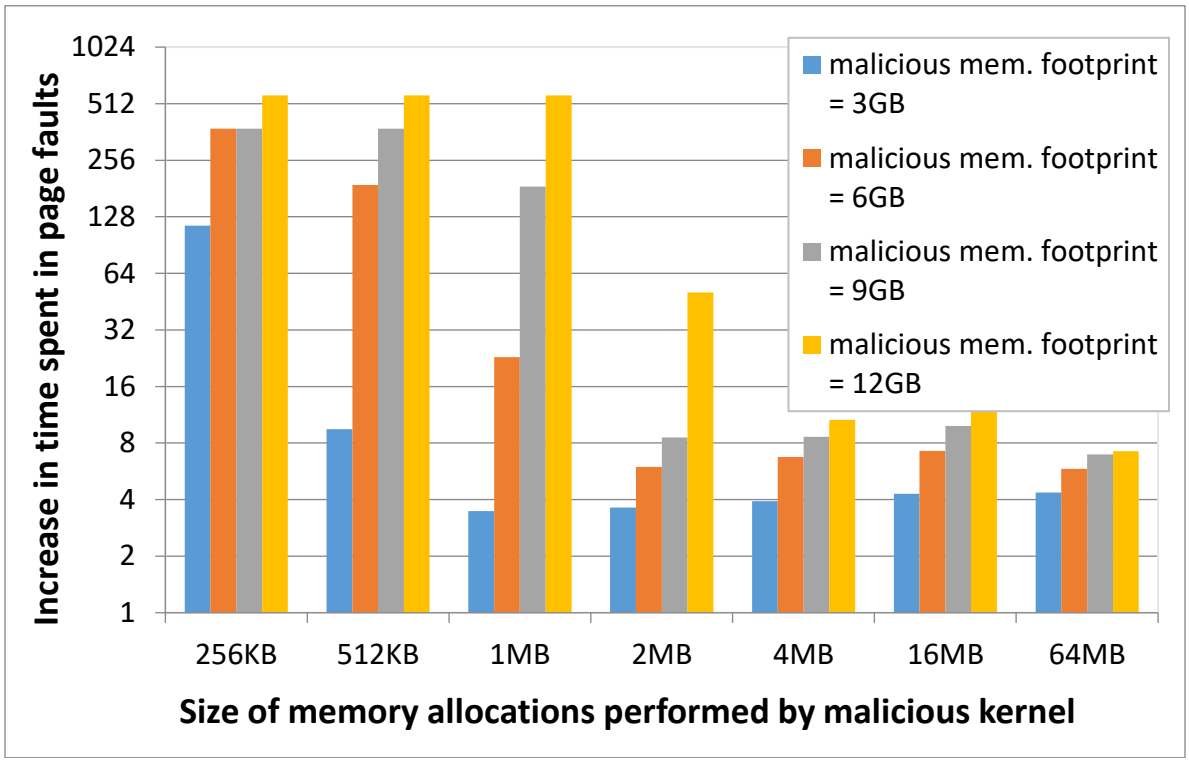

Figure 5.3: Bar chart showing the proportion of time spent servicing page faults across allocation sizes.

Thus, a greatly oversubscribed allocation (18GB) with a high trip count loop (50) keeping all pages in that oversubscribed allocation consistently brought the victim's execution to a standstill. We chose this configuration as our 'optimal' malicious kernel configuration for the remainder of our experiments.

The results from the first 3 runs (standalone, running alone in our proof-of-concept runtime, running alone in MPS) with the LoneStarGPU suite running solo (without any concurrent kernel companions) are shown below in Figure 5.4.

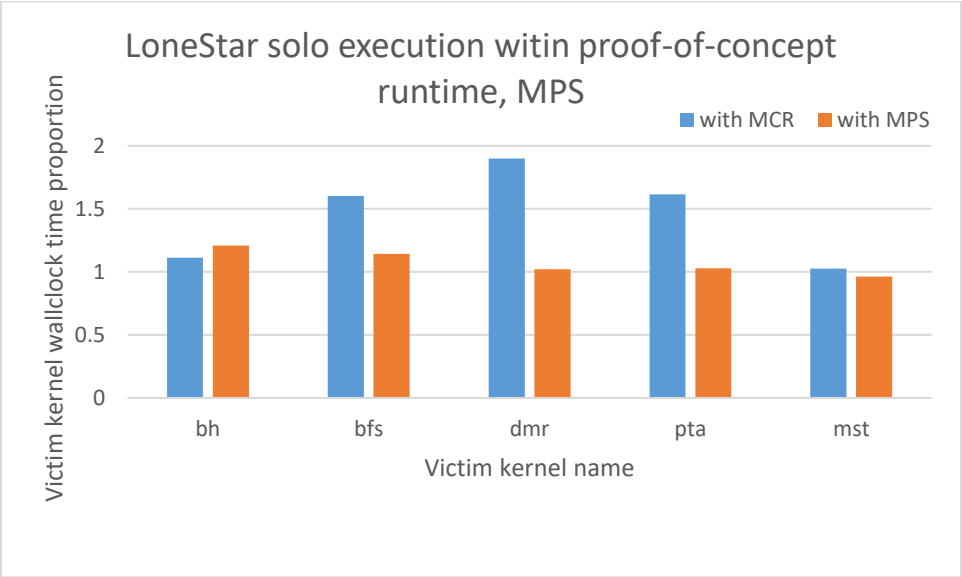

Figure 5.4: Bar chart showing the relative execution speed when LoneStarGPU is run within the proof-of-concept system and MPS Runtime Systems.

The vertical axis in Figure 5.4 above is the proportion of wall clock time consumed while running within the runtime systems relative to running standalone (no runtime used). From the bar chart above, we observe that the runtime systems we are using here are efficient enough for these applications since they do not contribute significantly to the overall wall clock time in relation to the magnitude of slowdown our malicious kernel does.

The chart below shows the relative slowdown of the benchmarks when co-run with a non-malicious kernel, bfs on the proof-of-concept runtime system.

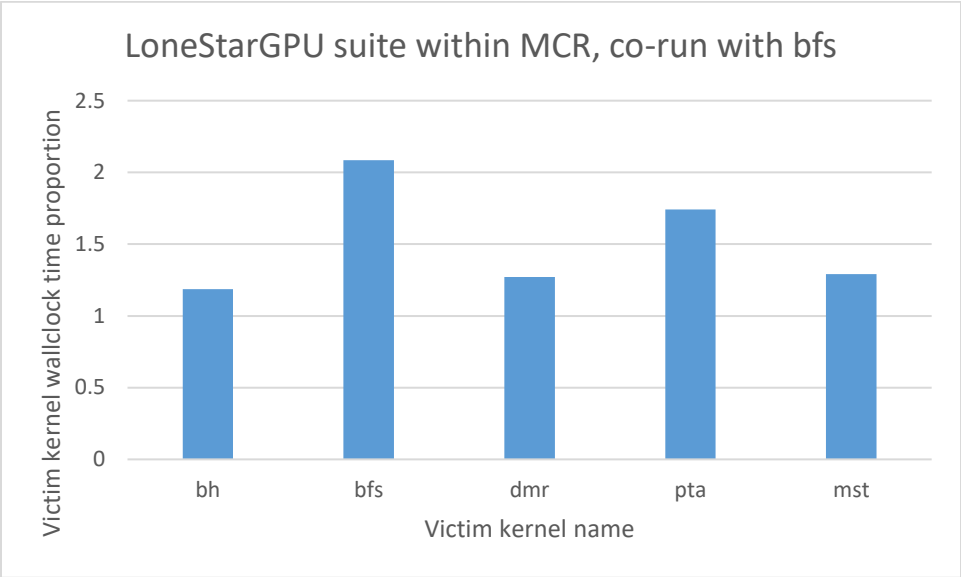

Figure 5.5: Bar chart showing the relative performance of the LoneStarGPU suite when co-run with an application (bfs) from the same suite in the POC system.

These results demonstrate that these kernels are good candidates for concurrent execution and our experiment, since none of them sees a major slowdown. This implies that their resource requirements can be accommodated easily even in the presence of a similar kernel. Thus, we can safely conclude that any additional slowdown we observe will be because of our malicious kernel, and not because of any secondary effects simply because we are concurrently executing them with another kernel.

As mentioned in the preceding section, we chose two representative configurations of the malicious kernel, a 6GB footprint spread across multiple arrays of 256KB each, and an 18GB footprint spread across multiple 1 MB arrays. With the latter configuration, we were able to achieve a complete DoS scenario. The smaller, non-oversubscribed configuration of the

malicious kernel was not able significantly slowdown the benchmarks. This is expected since there was no oversubscription and hence very little possibility of the malicious kernel forcing evictions of the victim's pages. Further, the slowdown is not as acute as with our simple vectorAdd kernel, because these programs involve much more computation than a simple addition that is in vectorAdd – making them less sensitive to memory performance. Each application of the LoneStarGPU suite has multiple kernels in each of these benchmarks and a CPU portion of the program also consumes time. That CPU portion is not directly affected by the slowdown on the GPU.

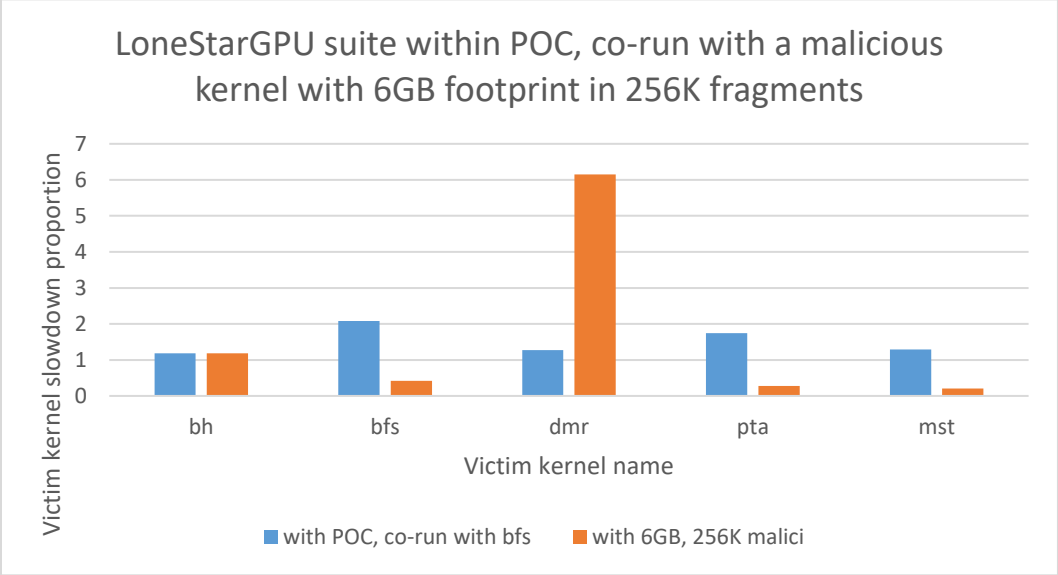

Figure 5.6: Chart showing the effects of a milder version of the malicious kernel on the LoneStarGPU suite as run in the POC system.

Motivation for an improved allocator

Armed with this data and our understanding of the Unified Memory System we set ourselves to the task of enhancing the existing Many Core Runtime to limit or prevent such a malicious kernel from affecting performance in this way. We noticed the most drastic result in

terms of slowdown was obtained when we went from a large continuous allocation to a fragmented allocation – essentially when we hinted the driver into using smaller pages.

For protecting against a malicious kernel that attempts to do this, we as runtime designers must prevent programs from being able to influence the page size heuristics in the driver in this way through the layout and pattern of their Unified Memory allocations. In the current context, keeping pages as large as possible would be the best thing to keep the negative effects of the oversubscription trigged evictions at bay. For a larger page size, a single page fault would bring in a larger chunk of memory. Thus, the cost of that single page fault would be amortized over multiple accesses – all of which would be subsequent page 'hits'. Consequently, the absolute number of page faults would decrease. While the number of bytes transferred would increase, the other per-fault software overheads would be much lesser in total since the over number of page faults would also be lower. This decrease is enough to prevent a complete DoS situation.

Pre-allocator Design

Based on this understanding, we implemented the idea described above and created a new memory allocator in the Many Core Runtime System, dubbed `cudaMallocManagedV2()`. This is essentially a wrapper around the original `cudaMallocManaged()` API, with one added mechanism. The main feature of this "V2" allocator is that it sets up pre-allocated Unified Memory pools for each Stream which are essentially large, continuous Unified Memory allocations themselves – the kind that the runtime would use large pages to store. The choice of maintaining independent pre-allocated pools for independent streams was made so that the effects of the heuristics in the kernel driver for a memory pool would be contained to affect a single stream (or application running in that stream), and not the entire system.

When an application calls `cudaMallocManagedV2()` for the first time, the wrapper code in turn triggers a `cudaMallocManaged()` to setup the memory pool. Subsequent allocation requests would not trigger any calls to the CUDA software stack, but instead get handled directly within the wrapper. The wrapper maintains a free-list for tracking allocations carved out of this pre-allocated pool. It also ensures that like the original function, it too returns addresses that are aligned to the texture alignment size – 512B for our device.

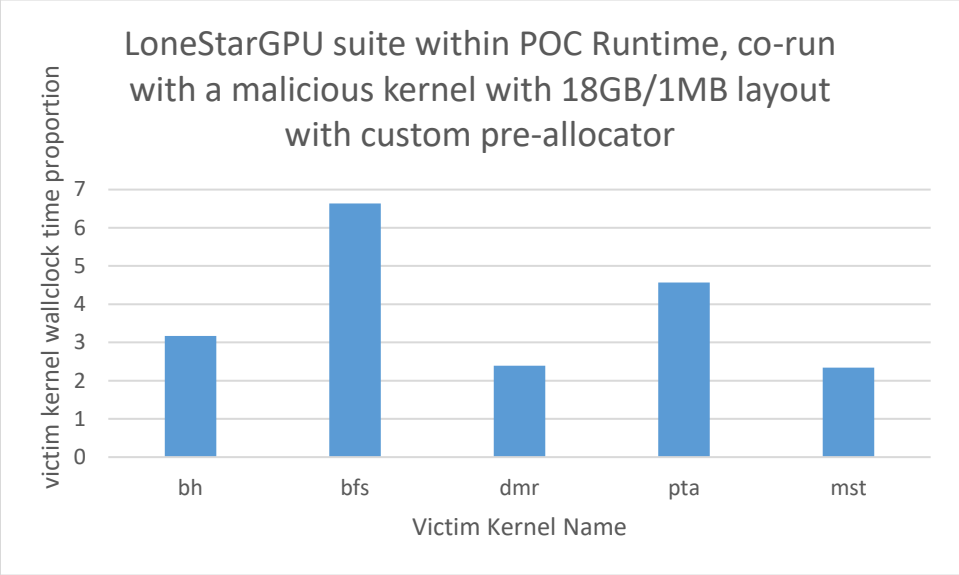

Figure 5.7: Bar chart showing the relative execution time of the LoneStarGPU suite when affected by the malicious kernel but using the pre-allocator.

The benchmarks experience varying proportions of slowdown compared to when they were run standalone, but no worse than 6X – which is still better than not being able to run at all. This simply designed allocator is enough to prevent a complete DoS.

Conclusion

We observe that is quite possible for a malicious kernel to abuse the demand paging feature of the Unified Memory system on Pascal GPUs and adversely affect the performance of other kernels when they are concurrently executing on the same device. The effect of the sizing of pages has a pivotal role in the severity of the effects of such malicious kernels. The main purpose of the pre-allocator was to prevent user programs from influencing the page size decision made by the UVM kernel driver, and instead keep it to a conservative larger page size to limit the damage from excessive page evictions. It bears noting that a large page size is not the final answer, because in workloads with mixed execution (CPU and GPU run concurrently on the same data), having larger pages than smaller ones forces larger pages to be frequently shunted between the processors. The benchmarks used here do not exhibit this execution pattern, but it is possible that other applications might. However, it does not take away from the relevance of keeping page sizes out of the applications' influence. A comprehensive solution thus cannot include simple runtime-based measures like our pre-allocator, but it must have active feedback from the UVM driver itself, as discussed in the next chapter.

CHAPTER 6: Future Work

The experiments carried out in this work pave the way for several possible compiler, runtime and driver-based solutions to tackle the issues studied here.

On the issue of memory protection, while using Unified Memory is an easy-to-deploy solution to fortify the protection around the working data set of a GPU kernel – we can do more. To definitively ensure that the data cannot be recovered, the commonly employed measure is to overwrite it. While the 5220.22-M Wiping Standard laid out by the U.S. Department of Defense [25] is definitely an overkill in this case, the basic idea of overwriting memory is still relevant here. We could introduce a compiler pragma to the programmer which they could use to mark certain allocations as 'sensitive' – say an array that stores the secret key involved in an encryption/decryption GPU kernel. An associated compiler pass would look for the presence of this pragma and inject `cudaMemset()` or equivalent calls to zero out those memory allocations once they are no longer needed (say, just before the programmer calls `cudaFree()`). Further, side-channel attacks that rely on predictable access latencies have also been proposed [26]. Such attacks could be foiled by the compiler padding data structures the programmer chooses to label 'sensitive' to introduce variation in the access latency – thus making it harder to glean any information about the underlying sensitive data structure.

The same solution could also be deployed as a runtime enhancement to the traditional `cudaFree()` call, by creating a "`cudaSecureFree()`" call that undertakes such data hiding/destroying measures in addition to its main purpose of executing the memory 'free' operation.

On the topic of providing some guarantee on the relative performance of concurrently executing kernels, current runtime systems currently enforce static mechanisms – such as the one

in MPS to limit the compute grid dimensions that can be launched on a particular stream [18]. Such static measures are definitely necessary, but they cannot contain the negative effects that the real-time execution of the kernel might create. The example of excessive page faults demonstrated in this work is one such negative effect that such static checks can't protect against. This is expected to some degree, since the runtime system has a very limited view of the actual execution of a kernel. The sphere of influence a runtime system-based solution can exert on running programs is limited to the Runtime API calls the program makes.

The right place to introduce mechanisms and policies to contain rogue or malicious kernels from executing in ways that compromise the performance of the system is the NVIDIA UVM Kernel Driver. It is the driver alone that has the best vantage point over kernels running on the hardware in terms of their Unified Memory performance. The driver could maintain access counters per page and be able to detect thrashing of pages (the same page(s) being evicted and allocated with high frequency) and recognize the presence of a possibly malicious kernel in the system. Further, it could also then adopt policies to tackle such rogue behavior since the driver is the one that decides which pages are evicted as part of servicing a page fault. The driver and runtime could expose controls to the user to mark certain kernel's pages as 'high priority' or 'sticky' so that they are not as vulnerable to such page-eviction attacks as this work demonstrated the average kernel to be. On that note, it might be interesting to readers that the UVM kernel driver does not[7] choose to evict pages allocated using the traditional allocator −

[7] While we cannot say for sure if it was intentional, the UVM driver maintains a separate free list and data structures to keep track of Unified Memory pages. The UVM driver does not seem to contain a global free list for all types of allocations. The free list for the traditional allocator is most likely separate and maintained in the closed source part of the driver.

`cudaMalloc()`. The latest CUDA Programming guide does indeed contain a fleeting admission of this nuance of the Unified Memory system. Thus, while it is lacking in memory protection, using the traditional allocator might be one way to avoid the vulnerability of perpetual page faults we have exploited here.

In terms of debugging and profiling the Unified Memory behavior of a program, the current toolchain does well to inform the user about the number of page faults the program saw throughout its execution, and the time that was spent servicing those faults. However, this high-level data alone is not sufficient for the programmer to tweak their code – especially if they are not completely familiar with the code base. With the introduction of instruction level preemption in Volta, the profiler is supposed to be able to support interactive debug of kernels. This implies that might also be possible to get line-by-line information about which memory references in the kernel tend to see the most page faults. The driver could also output a 'heat-map' of memory references to the program's allocated pages for each processor. This sort of feedback would be extremely helpful to the programmer to use the `cudaMemAdvise()` call more effectively. Current metrics are simply too broad for the programmer to get accurate feedback while tuning their program using this API. As with all computer systems, providing the right level of abstraction and control to the user or programmer that is helpful and relevant but not tedious or overly minimal is always a tricky thing to get right.

CHAPTER 7: Related Work

Study of memory protection and security on GPUs is an area that is garnering growing interest from the research community as well as ecosystem leaders like NVIDIA.

The vulnerability of the traditional allocator has been briefly studied in the 2013 publication from Di Petro et. al [27], but their investigation was limited to the traditional allocator, that too on very early GPU architectures. Our study not only evaluates Unified Memory in this regard, but also across multiple current architectures – Kepler/Maxwell and Pascal. The paper published by S. Lee et. Al [28] also exploits this very same vulnerability, but their primary focus is on how this vulnerability might be deployed in practice. They demonstrate visual proof of this vulnerability by partially recovering the frame buffer which they show still contained image data of the last image the GPU rendered. The GPU memory attacks described in the work of B. Di et. al [29] do achieve some of the same effects of 'data recovery' – but they adopt buffer overflows as their weapon of choice.

While there has been a lot published work [5] [6] [7] on the design of efficient GPU runtime systems and separately on the study of Unified Memory [30] [31] [32], we did not find any existing literature on the page-fault based attack we describe in this work. Closest published work to the pre-allocator idea proposed in this work is the micro-architecture based proposal of a memory management system for Unified Memory by R. Ausavarungnirun et. al [33]. In their study, they too recognize that varying page sizes can significantly affect the overall performance of the Unified Memory system. However, theirs is primarily a microarchitecture approach with the runtime playing only a supporting role, while we have limited our scope to the abstraction level a software runtime provides. Further, there has not been any work on incorporating such

performance guarantee mechanisms into a concurrent execution runtime or driver layer – both of which we propose in this work.

CHAPTER 8 : Conclusion

With the increasing adoption of NVIDIA GPUs as a mainstream computing platform, there is a fresh impetus to making them more scalable, cheaper and more energy efficient for applications in healthcare, gaming, virtual reality, machine intelligence, professional visualization etc.

With the expansion of its ecosystem and feature-set, like other computing platforms in the past to enjoy such attention and relevance – the GPU platform also becomes a potential target of malevolent interests. Industry leaders in this GPU computing revolution must strive to make their systems more robust and resilient to such malevolent interests.

This work represents a small step in that direction. It explores the possible vulnerabilities of the platform and how they may be exploited in the context of a concurrent execution runtime system. Along with proposing a page-fault based Denial-Of-Service attack, thus highlighting a potential vulnerability, we also prototype a simple solution that can be implemented at the runtime level to counter such an attack. In addition, we lay down the outline for other solutions both at the driver and runtime level that could be explored to improve the security measures on NVIDIA GPUs. The author hopes this work along with contributing to the existing body of knowledge in this area, shall also serve as inspiration for future efforts in GPU security.

REFERENCES

[1] – NVIDIA Developer Website CUDA-1.0 https://developer.nvidia.com/content/cuda-10

[2] – Amazon AWS EC2 Press Release https://aws.amazon.com/about-aws/whatsnew/2010/11/15/announcing-cluster-gpu-instances-for-amazon-ec2/

[3] – Microsoft Azure Discrete Device Assignment - Description and Background https://blogs.technet.microsoft.com/virtualization/2015/11/19/discrete-device-assignment-description-and-background/

[4] – S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. “Improving GPGPU concurrency with elastic kernels”. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’13, pages 407–418, New York, NY, USA, 2013. ACM

[5] – Zhong, Jianlong, and Bingsheng He. "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling." IEEE Transactions on Parallel and Distributed Systems 25.6 (2014): 1522-1532.

[6] – Mohammadi, Rasoul, et al. "A dynamic special-purpose scheduler for concurrent kernels on GPU." Computer and Knowledge Engineering (ICCKE), 2016 6th International Conference on. IEEE, 2016.

[7] – Yeh, Tsung Tai, et al. "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks." Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2017.

[8] – NVIDIA CUDA 6.0 Release Notes http://developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf

[9] – NVIDIA UVM Driver source code

[10] – NVIDIA DGX Press Release https://www.nvidia.com/en-us/data-center/dgx-2/

[11] – NVIDIA,"P100 GPU." Pascal Architecture White Paper (2016). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[12] –Sakharnykh, Nikolay. "Beyond GPU Memory Limits with Unified Memory on Pascal." (2016). At NVIDIA GPU Technology Conference 2017.

[13] - IBM Power Systems S822LC Press Release https://www.ibm.com/blogs/systems/ibm-nvidia-present-nvlink-server-youve-waiting/

[14] – NVIDIA CUDA Best practices guide https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[15] – Smith, Ryan “Preemption Improved: Fine-Grained Preemption for Time-Critical Tasks”, AnandTech, 2016. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10

[16] – NVIDIA, “Kepler architecture white paper.” https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[17] – Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. 2012. “A virtual memory based runtime to support multi-tenancy in clusters with GPUs”. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC '12). ACM, New York, NY, USA, 97-108.

[18] – NVIDIA MPS Runtime Documentation https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[19] – Butler, Michael, Kittisak Sajjapongse, and Michela Becchi. "Improving application concurrency on GPUs by managing implicit and explicit synchronizations." Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on. IEEE, 2015.

[21] – NVIDIA CUDA Driver API documentation https://docs.nvidia.com/cuda/cuda-driver-api/index.html

[22] – Burtscher, Martin, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs." Workload Characterization (IISWC), 2012 IEEE International Symposium on. IEEE, 2012.

[23] – Vastrad Adith, et. al “Porting LoneStarGPU to use Unified Memory”, 2018

[24] – NVIDIA Pascal Tuning Guide https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

[25] – National Industrial Security Program Operating Manual, United States Department of Defense http://www.dss.mil/documents/odaa/nispom2006-5220.pdf

[26] – Jiang, Zhen Hang, Yunsi Fei, and David Kaeli. "A complete key recovery timing attack on a GPU." High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. IEEE, 2016.

[27] – Pietro, Roberto Di, Flavio Lombardi, and Antonio Villani. "CUDA leaks: a detailed hack for CUDA and a (partial) fix." ACM Transactions on Embedded Computing Systems (TECS) 15.1 (2016): 15.

[28] – Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14).

[29] – Di, Bang, Jianhua Sun, and Hao Chen. "A Study of Overflow Vulnerabilities on GPUs." IFIP International Conference on Network and Parallel Computing. Springer, Cham, 2016.

[30] – Landaverde, Raphael, et al. "An investigation of unified memory access performance in cuda." High Performance Extreme Computing Conference (HPEC), 2014 IEEE. IEEE, 2014.

[31] – Li, Wenqiang, et al. "An evaluation of unified memory technology on Nvidia GPUs" Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on. IEEE, 2015.

[32] – Zheng, Tianhao, et al. "Towards high performance paged memory for GPUs." *High* Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. IEEE, 2016.

[33] – Ausavarungnirun, Rachata, et al. "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes." Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2017.

APPENDIX

APPENDIX A: Malicious Kernel source code

Included below is the source code for the malicious kernel proposed in Chapter 5:

```
 // Malicious Kernel
  __global__ void BadKernel(unsigned long int NumArrays, unsigned long
ALLOC_SZ_NUM_EL, unsigned long int** badArray){
      unsigned long int index = blockIdx.x * blockDim.x + threadIdx.x;
      unsigned long int stride = blockDim.x * gridDim.x;

      /* Operate on the right smallArray as decided by index */
      for (unsigned long int i = index; i < NumArrays; i+= stride){
          /* Access the badArray many many times */
          for (int loop = 0; loop < 50; loop++){
              for (unsigned long int j = 0; j < ALLOC_SZ_NUM_EL; j+=
stride_4K){
                  badArray[index][j] += 1;
              }
          }
      }
 }
```