Towards Demystifying Cache Interference on NVIDIA GPUs

Tyler Yandrofski
The University of North Carolina at Chapel Hill
tylerdy@cs.unc.edu

Jingyuan Chen
The University of North Carolina at Chapel Hill
leochanj@cs.unc.edu

ABSTRACT

Obtaining and bounding real-time tasks' worst-case execution times (WCETs) on NVIDIA GPU-based systems is challenging because the details of hardware and drivers are considered proprietary and lack documentation. This makes it difficult to predict the effects on WCETs of interference caused by contention for access to shared caches. This paper describes efforts to demystify, and thus possibly avoid, cache interference effects on NVIDIA GPUs by generating and measuring cache contention.

Keywords

GPUs, CUDA, cache interference

1. INTRODUCTION

Autonomous vehicles have received considerable attention across the real-time system community. Recent advances in computer-vision algorithms and graphics processing units (GPUs) have made fast and intelligent autonomous control systems possible. However, predictability in WCETs remains elusive and makes the certification of real-time safety in autonomous vehicles extremely challenging.

One major requirement for WCET predictability in safety-critical autonomous vehicles is the analysis of latency bounds. Such analysis requires careful consideration of interference from various sources, including cache, DRAM, bus, etc. However, on most NVIDIA embedded systems (currently the dominant platforms for autonomous vehicles) necessary information about the underlying hardware is hidden. Moreover, the memory architecture on GPUs could differ dramatically from that on CPUs, making traditional memory isolation techniques less suitable. Existing WCET estimations for GPU-based real-time systems are hampered by a limited understanding of cache and memory interference, potentially undermining the safety of these systems. The work described here addresses this issue by conducting a thorough investigation of cache interference on NVIDIA GPUs.

Organization This paper is organized as follows. We provide a brief overview of NVIDIA GPU and CUDA programming in Sec. 2 and discuss related work in Sec. 3. We then discuss our methods of efficient generation of extreme cache contention in Sec. 4 and conclude in Sec. 5.

2. BACKGROUND

2.1 GPU and memory architecture

We used the NVIDIA Jetson TX2 in this work. The Jetson line is marketed for "autonomous everything" and is commonly used in embedded applications. The TX2 has an integrated GPU, which shares DRAM with the CPU (as opposed to discrete GPUs with separate DRAMs common in gaming applications). The TX2 also contains a six-core heterogeneous ARMv8 CPU and a shared 8 GB DRAM. The GPU contains two streaming multiprocessors (SMs), each with a private L1 cache and 128 cores. The two SMs share an L2 cache which leads to contention and interference. Only two parameters of the L2 cache are documented – the total size (512 KB) and cache line size (128 bytes). We assume this cache is some variant of a set-associative cache [5].

2.2 CUDA basics

CUDA is a C/C++ extension for parallel programming developed by NVIDIA that allows programmers to execute *kernels*, programs to be executed on the GPU. Executing a kernel involves calling the kernel function with parameters expressing how GPU hardware threads will process the input data in parallel. Threads are organized into blocks, each of which is executed on a single SM. The number of blocks and number of threads per block is specified as parameters by the programmer. The threads in each block are scheduled on the SM in groups of 32 threads, called *warps*. Each thread in a warp executes the same instruction on different data (SIMD). When an instruction stalls on a memory access, the entire warp is context switched for a non-stalled one.

3. RELATED WORK

3.1 Managing cache interference on CPUs

Several works have provided methods to account for CPU cache and memory interference in real-time systems. [3] proposed a framework to profile memory access patterns of safety-critical tasks and a deterministic cache allocation framework to improve the predictability of the system. [6] introduced an analysis method that computes upper bounds of task delays due to memory contention. [7] presented a coordinated cache and memory bank coloring framework that reduces inter-task memory interference on multicore systems. These methods may not apply directly to GPU-based systems, where the computation models are fundamentally different and important architectural details are hidden.

3.2 Dissecting GPU hardware

[2] gave an extensive study on the hardware details for NVIDIA Volta architecture GPUs. [4] introduced a fine-

Table 1: Memory access time baselines (clock cycles)

Scenario	Mean	Minimum	Maximum	S.D.
Cache miss	519	461	1112	72
Cache hit	116	92	128	5

grained micro-benchmarking method to investigate the GPU memory hierarchies on the Fermi, Kepler and Maxwell architectural generations of NVIDIA GPUs. These works mainly measured unknown hardware parameters. In contrast, [1] conducted reverse engineering to unveil the details of the L2 cache and DRAM bank mapping functions on NVIDIA Pascal and Volta architectures.

While these works provide useful information about cache and memory behavior of NVIDIA GPUs, our work aims for a deeper understanding of how cache interference influences WCET predictability. We aim to understand not only how to prevent cache interference, but also how to produce it, so that we may bound WCETs of an arbitrary real-time task.

4. GENERATING CACHE CONTENTION

In this section, we discuss methods for generating extreme cache contention using an application-agnostic kernel that aims to evict all cache lines in the shortest possible time.

Motivation As discussed in the introduction, understanding cache contention effects on WCETs is necessary for the certification of safety-critical real-time tasks in autonomous vehicles. We also aim to inform real-time programmers about application patterns of memory access that lead to cache interference. Our goal is to design a cache-eviction kernel that effectively simulates extreme cache contention with minimal impact on GPU computation resources. We expect it to 1) evict enough cache lines to force most application memory accesses to go to DRAM and 2) run concurrently with any application, evicting efficiently according to the memory accessing behaviors of the system.

Eviction kernel design. The design must consider how kernel memory references result in cache accesses. The GPU L2 cache is used when kernels reference GPU global-memory virtual addresses. The addresses are mapped by hardware to physical addresses in the DRAM shared with the CPU. The physical addresses are then mapped by the cache to *sets* in the L2 cache, each of which can hold multiple cache line *ways*. The details of how physical addresses are mapped to cache sets are not publicly available. We initially assumed a linear mapping where the cache set is indexed by a set of k sequential low-order bits of the physical address [5].

We designed an eviction kernel to access GPU global-memory addresses that map to different L2 cache lines using *pointer chasing*. The kernel iterates through a global-memory array of unsigned integers equal in size to the cache. Each element in the array points to the index of the next element to be visited. Ideally, by visiting elements that map to the first word of each cache line, the kernel can evict every line in the cache.

4.1 Experiment designs

Our first step was to verify that the kernel that is successful in simple, ideal scenarios and simulations. We discuss the settings of these experiments below.

Time measurement. We measured times for memory accesses and kernel execution by reading the per-SM clock64 cycle-counting register. We stored the recorded times in the non-global per-SM memory shared by executing blocks and copied them to global memory later. We first recorded the memory access time for both warm and cold cache reads as baselines. We wrote a kernel that sequentially iterates through a cache-sized array and records the cold access times. Then the kernel iterates again and records the warm access times. Since the second pass could include cache misses, we discarded measurements in the second pass which fell into the cycle range observed to be associated with cache misses for the cold baseline. The observed mean time was 519 cycles for cold accesses and 116 cycles for warm accesses as shown in Table 1. Note that about 5% of cold accesses require more than 593 cycles. The reason for such occasionally large miss times is currently not clear. Potential explanations include DRAM row-buffer conflicts, bus contention, etc.

Run time configuration. To allow efficient eviction, we allocated multiple warps to the eviction kernel so that warp stalls for cache misses could be hidden by the warp scheduler. Also, to explore the effect of memory access patterns, two versions of the eviction kernel were written: one iterates through the arrays sequentially, the other randomly.

Potential problems. We assumed contiguous global memory maps to contiguous physical memory and physical addresses map linearly to cache sets. If these assumptions are violated, an array could contain more addresses that map to certain cache sets than to others. As a result, rather than accessing every cache set once for every cache line in it, the kernel would access some cache sets too few times (and others too many), leaving some cache lines not evicted. To explore this effect, we analyzed the cache coverage of arrays in different parts of memory using multiple pointer-chasing arrays.

4.2 Effective cache eviction

In this section, we explore methods to allow the eviction kernel to effectively evict all cache lines.

4.2.1 Warm cache experiments

Design. We first test our assumptions that contiguous virtual addresses are mapped to contiguous physical addresses and that the function that maps physical addresses to cache lines is a linear index. Under this assumption, each contiguous group of array elements aligned within a cache line should be mapped to a different cache line without any conflicts. Therefore, if a kernel iterates this array twice, all reads in the second pass should experience cache hits - full cache coverage.

We designed an eviction kernel that performs random pointer chasing over 8 arrays of 512 KB located at disjoint memory locations. For efficient execution, the kernel runs on 8 warps. The kernel iterates an array twice in a row, recording the time for each memory access in the second pass, before repeating for the next array. If our assumption holds, all access times should be consistent with the baseline for warm accesses. Figure 1 summarizes the results of multiple runs of the kernel for a sequential access pattern. A random access pattern yielded similar results.

Observations. The typical elapsed time for cache hits is in the range of 104 to 128 cycles[1], while the typical time for

[1]25% of accesses took 88 or 89 cycles because the nvcc compiler unrolls the loop into four PTX assembly sequences, the last of which is structured in a way that delays the memory access stall until after the access time is recorded

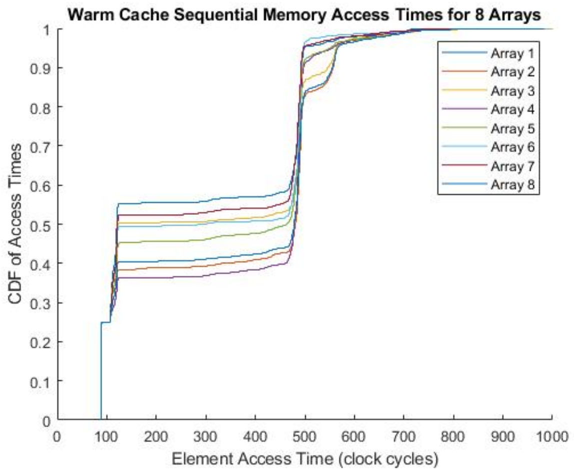

Figure 1: Cumulative distribution function plot of memory access times over 8 different sequential pointer chasing arrays.

cache misses is at least 450 cycles. These ranges are consistent with our baseline ranges. However, as shown in the figure, only 36% to 55% of the accesses are within the hit latency range, depending on the array. Thus, far from 100% of the cache was covered by accessing the cache-sized array - 36% in the worst case. The low coverage and variation in hit rates indicate our assumption does not hold. Non-contiguous physical memory or a non-linear hash function is used, which causes many physical addresses in the pointer chasing array to share cache lines.

4.2.2 Future work - Reverse engineering

Without low-level knowledge of the physical memory and cache mapping functions, it's almost impossible to achieve ideal cache coverage. Therefore, we plan to reverse engineer the mapping function of TX2's L2 cache. Obtaining the exact hash function would allow us to evict arbitrary cache lines with simple bit operations. Besides reverse engineering, we also aim to achieve optimal allocation of physical addresses for eviction, so that the eviction kernel may cover all cache lines without allocation of unrealistically large arrays or fragmented chunks of physical memory.

4.3 Concurrent eviction

In addition to evicting the entire cache when executed alone on the GPU, the eviction kernel must be able to stress a real victim kernel's memory accesses. We tested two methods of executing both kernels concurrently on the same GPU: 1) running in multiple processes and 2) running in different threads of the same process.

4.3.1 Multi-process experiment

Design. Our first experiment evaluated an application kernel's performance when executed concurrently with an eviction kernel in a second CUDA context (process). Note that in CUDA if multiple contexts share the GPU, only one context can access it exclusively at a given time. The victim application is a memory-intensive kernel that reads every integer in a cache-sized array (131,072 integers) sequentially and sums them. It runs with one block and one warp per block. The eviction kernel is the one from 4.2.1 but with

Table 2: Application execution times - multi-process

Opposing kernel	Mean (ms)	Max (ms)	S.D. (ms)
None	21.08	23.71	4.59
Compute 1x1	44.67	50.19	6.71
Compute 4x32	46.22	51.25	6.80
Eviction 1x1	44.69	50.18	6.71
Eviction 4x32	46.24	52.49	6.80

a random access pattern. Two different thread layouts are used: 1 block-1 warp per block and 4 blocks-32 warps per block (the max threads available to the TX2). We ran the application in three scenarios: alone as a baseline, concurrently with a "compute-only" kernel in a second process, and concurrently with a true eviction kernel in a second process. The "compute-only" kernel simply executes an infinite loop without accessing memory. In this case, we can measure the effect of multi-process concurrent execution separately from the effect of cache interference.

Observations. Table 2 summarizes 4096 samples of the application kernel's execution times per scenario. The "None" scenarios mean the application runs alone as a baseline. While the execution times were much higher with the compute-only kernel than with the baseline, the compute-only and eviction kernel results are similar. This indicates that the application process experiences increased execution times when executing concurrently with another process, but that the eviction kernel's memory accesses had little effect.

Discussion. Both results can be explained by considering that multi-process execution in CUDA is achieved by time-sliced execution, rather than real concurrent execution. Due to time-slicing and context switches, the execution times will approximately double when the application kernel shares the GPU with other contexts, as it did with the compute-only kernels (2.12x and 2.19x increase compared to application-only). Only memory accesses that re-establish the application's working set at the start of its time slice were susceptible to eviction-triggered cache misses. This working set interference was overwhelmed by the 2x time-slicing effects.

4.3.2 Single - process experiment

Design. Since time-slicing effects limit interference between processes, we next explored cache interference between kernels in a single context (process). We executed a victim application that randomly accesses elements in an array of cache size (131,072 integers) as a thread (*pthread*) in the process. Another thread in the same process ran the eviction or compute-only kernels from 4.3.1, but with a 2 block-32 warps per block thread layout. By allocating 64 warps to the eviction kernel (as opposed to 1 application warp), we minimize the effect of stalls due to cache misses since many other warps are available to schedule in place of one that stalls. Additionally, each kernel ran on a dedicated SM, avoiding interference effects caused by sharing cores with the victim. As before, we ran the application alone, with a compute-only kernel, and with a true eviction kernel.

Observations. Measurements of 4096 samples for each setup are summarized in Table 3 in the entries labeled "R" for random application kernel (the sequential "S" entries are described in 4.4.1). "None" scenarios correspond to the application kernel running alone. This time, the application-only measurements are similar to those for the compute-only se-

Table 3: Application execution times - single-process

Scenario	Mean (ms)	Max (ms)	S.D. (ms)
None-R	30.72	33.27	0.04
None-S	17.15	19.49	0.04
Compute-R	30.97	33.15	0.04
Compute-S	17.36	19.43	0.03
Eviction-R	64.59	64.81	0.06
Eviction-S	23.25	23.35	0.03

tups, a sensible result since there were no competing warps on the application's SM. This indicates that the execution times were not affected by GPU scheduling policies so any increase in application execution times can be attributed solely to the memory accesses of the eviction kernel.

Discussion. The results show that significant cache interference between kernels in the same CUDA context is possible, although our methods in this experiment are impractical for two reasons. First, allocating many warps to the eviction kernel and few to the application is unrealistic, especially since an entire SM was dedicated to the eviction kernel. More work is needed to determine a balance of execution resources that creates enough cache interference for accurate WCET measurement without unduly restricting application performance. Second, we executed the kernels in a single process by running both programs as threads, which is not viable for every potential victim application since it requires significant modification to process-based applications.

4.3.3 Future work - MPS

We plan to use CUDA Multi-Process Service (MPS) to solve the second problem without introducing the time-sharing effects observed in 4.3.1. MPS is an implementation of CUDA for truly concurrent execution of multiple processes' GPU kernels within a single context managed by the MPS server. Future work is required to implement the experiments from this section using MPS and confirm that sufficient cache interference is possible using this approach.

4.4 Eviction-application interaction

In addition to ensuring full cache coverage and efficient concurrent execution, it's important to consider how characteristics of application memory access patterns might impact cache interference. Our next experiment considers sequential application data accesses.

4.4.1 Sequential vs. random application accesses

Design. Our scenario was the same as in the previous experiment (1 warp application kernel and 64 warp eviction kernel), but with the application making sequential accesses rather than random ones.

Observations. Table 3 compares the results for this experiment (sequential application labeled with "S") and the previous one (random application labeled with "R"). Note that the execution times for sequential access scenarios are much smaller than those for random access, both with and without the eviction kernel executing.

Discussion. The observed results can be explained by the presence of a read-ahead effect in sequential memory accesses. A cache line that has already been buffered in the SM due to a previous read will have a lower access time than if it had required a cache or DRAM access. This effect would reduce cache interference for any sequential memory access pattern with read-ahead.

4.4.2 Future work

In future work, we will evaluate the effects of other memory access patterns on the interaction between eviction and victim kernels. For example, applications with smaller working sets may prove to be difficult to evict quickly enough to create interference. Certain distributions of application data in physical memory may make interference easier by concentrating data in locations that map to certain cache sets. The eviction kernel pattern may matter as well, for instance evicting an entire cache set at a time versus rotating between lines from different cache sets. It may also be possible to focus eviction efforts on cache sets that are high-frequency for a specific application.

5. CONCLUSIONS

In this paper, we discussed challenges in generating extreme GPU cache interference and presented our current experiments and potential future work to achieve powerful cache eviction kernels. By gaining a more detailed low-level understanding of the GPU cache architecture and available concurrency options, a more profound analysis of the impacts of cache interference on GPU-based systems, including case studies on real workloads, will be possible in the future.

6. REFERENCES

[1] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019.

[2] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[3] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

[4] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[5] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[6] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[7] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692. IEEE, 2013.