Contention Tracking in GPU Last-Level Cache

Javier Barrera†,‡, Leonidas Kosmidis†,‡, Hamid Tabani†, Jaume Abella†, Francisco J. Cazorla†
†Barcelona Supercomputing Center, Barcelona Spain ‡Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
{javier.barrera, leonidas.kosmidis, hamid.tabani, jaume.abella, francisco.cazorla}@bsc.es

***Abstract*—The Last-level cache (LLC) is one of the main GPU's shared resources that contributes to improve performance but also increases individual kernel's performance variability. This is detrimental in scenarios in which some level of performance predictability is required. While predictability can be regained by deploying cache partitioning (isolation) mechanisms, isolation negatively affects performance efficiency. This work shows that not partitioning the LLC and providing the ability to track the contention that kernels generate on each other allows them to share LLC space, hence increasing efficiency, while the system designer obtains a clear view of how each kernel affects each other in the LLC so as to balance performance and predictability goals. In this line, we propose GPU demotion counters (GDC), a low-overhead hardware mechanism to track contention that kernels generate on each other in the shared LLC.**

I. Introduction

We are witnessing a trend towards allowing kernels from different applications to share GPU resources [13]. For instance, NVIDIA Multi-Process Service (MPS) allows kernels from different applications to run in different Streaming Multiprocessors (SMs) simultaneously, and hence, share the LLC. While resource sharing allows increasing the aggregated performance of all kernels, per-kernel performance might suffer high variability depending on the usage of resources made by co-runner kernels ultimately causing loss of performance determinism [18]. This is detrimental in scenarios in which some level of quality of service (QoS), i.e. performance predictability, is required.

The LLC is one of the main sources of performance improvement and potential QoS degradation [18]. To regain predictability, hardware [4], [10], [14] and software [7], [9] isolation mechanisms are leveraged. Both restrict different applications to use a fixed subset of the LLC space determined beforehand, preventing the eviction of each others' data. However, while cache partitioning allows regaining predictability, it reduces resource usage since the LLC space not used by an application is not available to its co-runners.

While in hard real-time systems cache partitioning is the most natural choice, as predictability is prioritized over average performance, many other application domains demand some QoS levels that call for a better balance between performance and predictability. In those scenarios, it would be possible to allow different applications to share the LLC as long as the hardware provides run-time system information about how applications affect each other in the LLC. This information can be used by the system designer to propose run-time mechanisms to achieve QoS and performance goals, ranging from changing application schedule [18] to temporarily restricting the frequency at which an application is allowed to access the LLC to contain its impact on its co-runners [15].

In this line, for embedded GPUs, we propose and evaluate GPU demotion counters (GDC), a technique that allows tracking the contention that kernels in a GPU generate on each other in the LLC. GDC allows breaking down the LLC cache misses suffered by a kernel among its co-runner kernels. That is, GDC allows ascribing a percentage of the LLC misses to each of its co-runners, hence providing key information about the contention that kernels generate on each other in the LLC. This is fundamental during testing for timing violation *detection* and *correction* to single out its specific causes [11]. We illustrate how GDC applies to a demand-based replacement policy like Least-Recently Used (LRU).

II. Tracking LLC Contention in GPUs

The use of a shared LLC, the L2, is common in GPU subsystems in MPSoCs. The LLC is dynamically shared among SMs, acting as the first coherence point.

Common replacement policies, like LRU, have the stack property [8]. For LRU, each cache set can be conceptually seen as an LRU stack with lines sorted based on their last access cycle. The first line of the LRU stack is the MRU, position 0, and the last is the LRU, position W-1, where W is the number of LLC ways. The closer a cache line in a set is to the LRU position, the more likely it can be evicted by following accesses.

GPU LLC contention tracking techniques aim to break down the number of LLC misses (M_i^{cont}) suffered by the kernel under analysis (KUA or K_i) when it runs in a workload with other contender kernels, or CKs. In the following sections we assume that one kernel from a different application runs per SM. One is the KUA, while the rest are CKs.

A. Per-Line Owner Bits (PLOB)

Cache lines comprise control bits that provide information such as whether the line is valid and some bits to implement the replacement policy (e.g. LRU). PLOB adds $log_2|SM|$ bits per cache line to store the information regarding the owner (ow) of that line, that is, the ID of the SM from where the data in that line was last loaded or stored. A similar solution has already been proposed in the context of CPUs keeping the application ID rather than the kernel (or core) ID, but for similar purposes [18].

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB-C21 and IJC-2020-045931-I funded by MCIN/AEI/ 10.13039/501100011033 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773).

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/ICCD56317.2022.00021

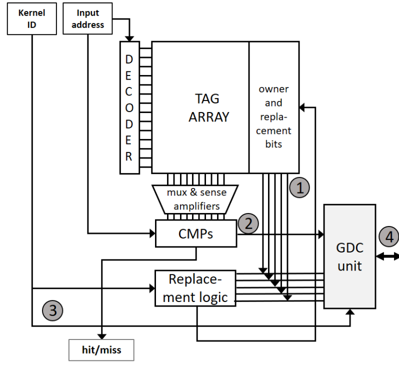

Fig. 1: GDC integration in the LLC

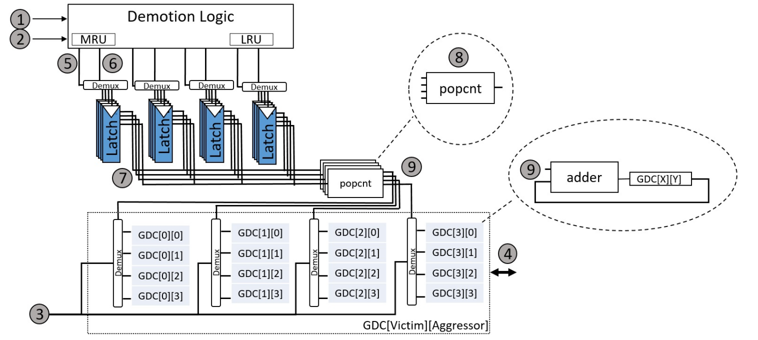

Fig. 2: GDCU diagram for a 4-way LLC and a 4-SM GPU.

On a miss, by comparing the PLOB of the line to be evicted with the ID of the kernel generating the access that causes the eviction, PLOB determines whether a kernel evicted data from another kernel. To store this information, PLOB also requires $|SM| \times |SM|$ event monitor registers to keep track of the number of times a given kernel evicted data from another one. These event monitors are referred to as inter-kernel eviction (ike) monitors. That is, when K_j evicts a piece of data in the LLC from K_i, ike_i^j is incremented. With this, KUA's LLC contention is broken down as $\delta_i^j = M_i^{cont} \times ike_i^j / ike_i$, where ike_i is the number of LLC ike suffered by K_i and δ_i^j, the K_i's LLC misses ascribed to K_j. Recall that M_i^{cont} is the number of LLC misses suffered by K_i.

III. GDC

The main principle behind GDC is that, right after the KUA accesses address $@A$ (for simplicity in the discussion we assume that each address is mapped to a different line), $@A$ is promoted to the MRU position. In order for this address to be evicted by other accesses, either from the KUA or from CKs, $@A$ needs to be progressively shifted from the MRU to the LRU and then be evicted. Hence, $@A$ suffers a sequence of *demotions* from the MRU position to the LRU+1 position.

GDC builds on the fact that an access to a line that hits in position k of the LRU stack causes it to be promoted to the MRU position, with all lines between MRU and $k-1$ being demoted one position each. Also, an access that results in a miss causes all lines in the set to be demoted by one position. That is, on every access to a line by a kernel, GDC tracks the impact it has on the LRU distance of the lines of other kernels, so, how much those lines are demoted. Hence, by tracking 'demotions', GDC accurately estimates how much each kernel contributes to the eviction of a line in the LLC.

A. Implementation

GDC is implemented via the GDC unit (GDCU), whose main goal is to track the number of demotions that the lines of a given kernel suffer, from itself or other kernels. To that end, the GDCU is integrated into the LLC as shown in Figure 1, which only shows the TAG array of the cache since the data array is not needed to describe the GDCU.

The GDCU builds on PLOBs, so, every line is tagged with the kernel ID bits. On every access to the cache, the index bits of the address are used to drive the LLC decoder that selects a cache set. The tag information is sent to a comparator to determine whether the access is a hit or a miss. Also, the LRU bits are sent to the replacement logic, which updates them and writes them back in the corresponding set. While it is not mandatory for GDCU functioning, we assume an implementation in which replacement bits are kept together per set, in the form of a sorted list of way identifiers from the MRU to the LRU way. For instance, for a 4-way LLC where the MRU for a given set is way 2, followed by way 0, way 3 and way 1 (so way 1 is the LRU), the replacement bits would be the concatenation of 10_b, 00_b, 11_b, and 01_b, sorted from MRU (left) to LRU (right). Along with the way identifiers, the owner bits (kernel IDs) are stored. We refer to both together, LRU stack bits and owner bits, as replacement bits.

The GDCU has three main inputs, the replacement bits ①, the position in the LRU stack on which the hit was detected ②, and the ID of the kernel performing the access ③, see Figure 1. These three inputs are used to update the GDCU demotion counters (or simply counters). The associated logic of the GDCU can provide enough throughput so that it processes LLC accesses at the same frequency as the LLC itself. If the GDCU latency is larger than the LLC inter-access latency (i.e. the shortest time elapsed since one access is processed until the following access is processed), then the GDCU logic can be easily pipelined accordingly. This only causes the counters to be updated a few cycles after the actual access happens. This delay has no impact as the GDCU reading and resetting of counters is done every dozen thousand or even a hundred thousand cycles (e.g. at task boundaries) via a dedicated input/output port ④.

More in detail, the *demotion logic* in the GDCU (Figure 2) needs the kernel IDs (KIDs) owning the lines of the accessed cache set sorted from MRU to LRU (note that the KID of an active kernel varies from 0 to $|SM|-1$). The most natural way to obtain those IDs is via the replacement bits ① that, as explained before, include a list of way identifiers along with their corresponding KIDs; and the LRU position where the hit happened. In case of a miss, a different value is coded in ②. The demotion logic drives the demotion activation bit to a demultiplexer ⑤. This signal is 1_b if that position of the LRU stack has been demoted and vice versa. Recall that, with LRU,

TABLE I: Main configuration of the simulator platform

CPU Configuration	
Number of Cores	4 Armv8
L1 Inst. Cache	32KB 2-way write-back
L1 Data Cache	64KB 2-way write-back
L2 Shared Cache	2MB 8-way write-back
Main Memory Size	8GB
GPU Configuration	
Number of SMs	4
L1 Inst. Cache per SM	4KB 4-way, 8 sets 1 bank, 128 bytes per line
L1 Data Cache per SM	24KB 48-way, 4 sets 1 bank, 128 bytes per line
L2 Cache (LLC)	512KB 16-way, 256 sets 4 banks, 128 bytes per line 4 sectors per line, 32-bytes each

on a hit, all lines between the MRU position (included) and the line hit (excluded) are demoted one position towards the LRU position, and hence their associated enable signals are asserted. Meanwhile, on a miss, all lines are demoted by one position so the enable bit of all LRU stack positions in the demotion logic is asserted.

The control input to the demux is the KID, that for our example with 4 SMs has two bits ⑥. Note that entries ① and ② correspond to the status before the tag data in the cache set is updated as a result of the access being performed by ③.

The output of the demux are $2^{|SM|}$ signals (one per kernel), with signal X for the KID kernel activated if such line has been demoted. The output ⑦ can be directly driven to the *popcnt* (population counter) block or latched (as shown in the figure) to ease pipelining the GDCU. There are $|SM|$ popcnt blocks, one per KID. Popcnt X is inputted with the output line X of each demux block, so that each popcnt has as many input lines as cache ways has the LLC, i.e. 4 in our case ⑧.

In our case, the output of the popcnt$_X$ is the number of demotions kernel K_X has suffered ⑨. This information is driven to the counters (GDCs). Input ③, i.e. the KID of the kernel performing the access (e.g., Y), is used as the control signal to activate only the counters related to K_Y, so the output of the $2^{|SM|}$ popcnts increase the value of counters $GDC[*][Y]$. Hence, in Figure 2 only one row of counters is incremented on access. We assume one adder per column (i.e. victim kernel) to update GDCs in parallel. Note that simple adders can be implemented since the maximum number of demotions is very low (i.e. up to the number of cache ways), and hence a regular adder is used for those bits, and only carry propagation logic for uppermost bits.

IV. Evaluation

A. Experimental framework

We use the Gem5-GPU [12] simulator that integrates Gem5 [2] to model the CPU and the memory subsystem and GPGPU-sim [1] to model the GPU. This infrastructure is considered among the most sophisticated and cycle-accurate simulators in both academia and industry. We have modeled an architecture in which the CPU and the GPU share the main

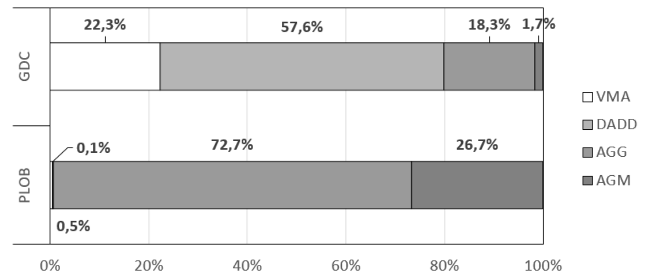

Fig. 3: Real Scenario

memory via an interconnection network, similar to embedded GPU devices such as NVIDIA AGX Xavier, see Table I.

We use several representative GPU benchmarks (basic operations) that are commonly used in machine learning libraries, which in turn, are used for many operations of autonomous driving and ADAS software, from perception and detection to planning and control: Matrix Multiplication (MM), Matrix Transpose (MT), Matrix Transpose Multiply (MMT), and Vector-multiply-add (VMA). For instance, matrix multiplication is a central element of YOLOv3 machine learning library [16] and radar applications [5], [17], and has been shown to account in some scenarios for 67% of YOLO's execution time [3].

We also used a set of basic operators with different data types and precisions. In particular, we use vector addition with integer long and with floating-point double-precision (LADD and DADD) and vector multiplication and division with long and double floating-point types (LMUL, DMUL, LDIV, and DDIV). All these operators are the building blocks for other basic functionalities in machine learning libraries and radar applications in automotive applications.

We also develop a set of aggressive benchmarks (AGG1 and AGG2), which put different degrees of high pressure on the LLC of the GPU.

B. Analysis of a real workload (VMA, DADD, AGG1, AGG2)

VMA (the KUA) and DADD have a footprint of 25% and 50% of the LLC space respectively. AGG1 performs sustained misses across all cache sets, and AGG2 performs sustained misses across a reduced number of sets. When executed together, VMA and DADD end up keeping most of their contents in LLC performing many hits and reusing data quickly. AGG2 performs sustained misses causing a small LLC miss increase on the KUA since most of the contents of the few sets accessed by AGG2 belong to AGG2 itself. AGG1 misses in all sets but at a much lower frequency than VMA and DADD accesses since AGG1 experiences long memory latencies and VMA and DADD short hit latencies. Hence, despite each AGG1 access pushing all lines in the accessed set towards the LRU position, it accesses the LLC seldom, while VMA and DADD access it at high frequency. As a result, VMA pushes its own lines little since its reuse distance is relatively short, whereas DADD pushes VMA lines much more due to its larger reuse distance.

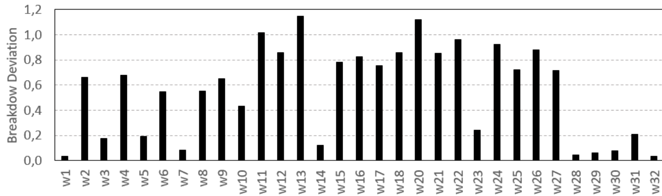

Fig. 4: PLOB breakdown deviation for 32 randomly generated workloads w.r.t. GDC's contention breakdown.

GDC reflects this behavior accurately with AGG2 being ascribed a small fraction of VMA's misses (1.7%) and AGG1 a percentage smaller (18.3%) than that of DADD (57.6%) that is the kernel affecting the most VMA. Also, VMA self-generates some of its misses (22.3%).

Instead, PLOB largely overestimates the impact of AGG1 and AGG2, which are the ones missing in the LLC, with AGG1 ascribed 72.7% of the misses of the KUA and AGG2 26.7%.

C. Wider Result Set

Next we show how off PLOB is from GDC as a measure of its accuracy. To that end, we define the metric *workload breakdown deviation (wbd)* that builds on the square root of the distance between two points in an N dimensional space.

For a workload w with N kernels, the wbd between GDC (A) and PLOB (B) is defined as $wbd_w^{A,B} = \sqrt{\sum_{j=1}^{N}\left(p_j^A - p_j^B\right)^2}$, where p_j^X is the percentage of L2 misses of the KUA ascribed to K_j by technique X so that the higher the value of wdb, the worse it behaves with respect to GDC and hence, breaking down LLC misses of the KUA.

Figure 4 shows the wbd for 32 randomly generated workloads. As it can be observed, the wbd of PLOB with respect to GDC can range from values as small as 0.03 (w1) to 1.15 (w13). This wide deviation range is caused by the characteristics of the workloads: PLOB tends to perform better when the different kernels in the workload have similar memory footprint and access frequency, and vice versa. For instance, w13 comprises more diverse kernels, with CK2 having the lowest access frequency and the KUA the highest. CK2, due to its low access frequency, is unable to reuse LLC data and performs most of the evictions, and thus, is ascribed as the main responsible for KUA's misses by PLOB. However, this disregards demotions performed by the other three kernels, which are ignored by the PLOB. Overall, PLOB is subject to pathological eviction scenarios in which the kernel causing the eviction is just the one causing the last demotion, i.e. from the position LRU to LRU+1, while in reality other kernels are the ones really pushing the line towards the LRU position.

V. Related Work

Some authors propose a similar solution to PLOB [18], CacheScouts, to track contention. CacheScouts keep the application (owner) ID of each cache line, instead of identifying the actual kernel, as in our case for GPUs. However, since the application ID size is potentially high, the authors propose sampling contention in a few sets to reduce costs. In our work, we consider a simpler implementation of PLOB where the kernel ID is tracked, which requires a few bits. Hence, we do not rely on sampling. Building on [18], other authors also from Intel, describe the Cache Monitoring Technology (CMT) in the Intel Xeon Processor E5-2600 v3 Product Family [6]. CMT allows monitoring of LLC cache occupancy by tagging a subset of cache lines in the shared L3 (LLC) as in [18]. This allows software to track cache occupancy per core.

VI. Conclusions

Allowing applications to share the LLC and providing hardware support so that the system designer can get a clear insight on how applications affect each other in LCC, allows balancing QoS and performance, and is key during testing stages to single out the specific causes of timing violations and apply the appropriate corrections. We propose GDC (GPU Demotion Counters) that allows a tightly tracking of the number of evictions that kernels in a GPU generate on each other in the LLC. We show that, by focusing on how kernels demote each others' data rather than only on direct inter-kernel evictions, GDC improves other LLC miss breakdown techniques based on existing and proposed hardware, while incurring low implementation overheads in the LLC.

References

[1] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS 2009*, 2009.
[2] N. Binkert et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.
[3] F. dos Santos et al. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *DSN-W 2017*.
[4] Freescale semicondutor. e6500 Core Reference Manual. https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf, 2014. E6500RM.
[5] J. Gamba. *Automotive Radar Applications*. 2020.
[6] A. Herdrich et al. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *HPCA 2016*.
[7] S. Jain et al. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *RTAS 2019*, pages 29–41. IEEE, 2019.
[8] R. L. Mattson et al. Evaluation techniques for storage hierarchies. *IBM Syst. J.*
[9] S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 2017.
[10] M. Moretó et al. Dynamic cache partitioning based on the MLP of cache misses. *Trans. High Perform. Embed. Archit. Compil.*, 2011.
[11] J. Pérez-Cerrolaza et al. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4):79:1–79:38, 2020.
[12] J. Power et al. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Computer Architecture Letters*, 2014.
[13] R. Pujol et al. Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier. In *ECRTS 2019*.
[14] M. K. Qureshi et al. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 2006*.
[15] M. Slijepcevic et al. Time-analysable non-partitioned shared caches for real-time multicore systems. In *DAC 2014*.
[16] H. Tabani et al. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *ISORC 2020*.
[17] L. Teschler. The basics of automotive radar, 2019. https://www.designworldonline.com/the-basics-of-automotive-radar/.
[18] L. Zhao et al. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT 2007*.