2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3503-2697-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/COMPSAC57700.2023.00022

L2 Cache Access Pattern Analysis using Static Profiling of an Application

Theodora Adufu* and Yoonhee Kim †
* †*Department of Computer Science*
Sookmyung Women's University
Seoul, South Korea
Email: *theoadufu@sookmyung.ac.kr, †yulan@sookmyung.ac.kr

***Abstract*—Cache management is a significant aspect of executing applications on GPUs. With the advancements in GPU architecture, issues such as data reuse, cache line eviction and data residency are to be considered for optimal performance. Frequency of data access from global memory has significant impacts on the performance of the application with increased latencies. However, the L2 cache data residency feature by NVIDIA promises to reduce the overheads associated with frequent data accesses. Through the information extracted from static profiling analysis, we quantitatively analyzed the frequency of data reuse by threads to determine whether an application has frequent data accesses or not. We also estimated the size of access policy window from which persistent data should be cached to avoid stalling of warps. Also with our proposed approach, we observed that L1 cache load throughput increased by 2.75% for GEMM, 0.33% for 2DConv St and 0.46% for 2DConv Large respectively as data was resident in the L2 cache.**

***Index Terms*—L2 Residency Control, Coalescing Graph, Static Profiling, Frequently Accessed Data**

I. Introduction

Recent advancements in computational science, specifically the improved capabilities of Graphics Processing Units (GPU), are mainly driven by compute-intensive workloads, such as Machine Learning (ML) and High performance computing (HPC) applications. In HPC and cloud environments [1] [2] [3] [4], GPUs remain a necessary computing resource for the efficient execution of diverse workloads.

Modern GPU architectures offer users exponential GPU capacity for the execution of diverse applications. However misaligned memory accesses, poor data locality in the cache memory, high miss rates and cache thrashing can be costly on performance. GPUs can hide memory access latencies with computation as multiple threads execute the same instruction in parallel however this can be further improved by managing caches efficiently.

NVIDIA, with it's Ampere architecture, offers a new feature that allows the programmer to manage data persistence in a defined portion of the L2 cache through the use of APIs [5]. Defining and maintaining persistent data in the set-aside region ultimately enables higher bandwidth and further lowers access latency to device memory.

However, applying residency control in the L2 cache is a laborious task if the programmer desires any significant results. The programmer is expected to define a set-aside portion, understand the data reuse frequency of the application and also consider the same in concurrently running applications. Fine-tuning the L2 cache residency configurations add to the programmer's burden of optimizing the performance of an application.

This paper investigates the use of static profiling using PTX code to determine the L2 cache residency control configurations as a means of optimizing memory and thus improving performance. Our research work focuses on determining the size of the data reads to be considered in the access window (num_bytes). Through this study,

- We determine the memory access patterns of applications through static profiling
- We determine the data access type of the application using quantitative analysis on the frequency of data reuse by threads and classify the applications into one of two groups
- We estimate the size of the access policy window for each application and use the estimated value to tune the L2 residency in the application.
- We evaluate the approach with selected applications.

The proposed approach eliminates the time spent in dynamically profiling the application during each run and is independent of different GPU architectures. Our work promises to optimize performance in concurrently running applications in spatially shared environments.

The rest of the paper is organized as follows: in Section 2, we briefly describe the background and motivation for this study. In Section 3, we give details of the proposed static profiling approach to determining the configurations for the L2 cache. We then go on to describe the configurations for the L2 cache residency introduced by NVIDIA in Section 4. We present the quantitative results of our study in Section 5 and discuss some related works in Section 6. We conclude the paper in Section 7.

II. Background and Motivation

According to Walden et al. [6], the data layout of applications like the sparse matrix in memory, along with relatively small blocks, poses a challenge for GPU architectures to utilize the memory bandwidth effectively. Threads from such applications require data accesses to different memory locations

†Corresponding Author: Sookmyung Women's University, Department of Computer Science, yulan@sookmyung.ac.kr

979-8-3503-2697-0/23/$31.00 ©2023 IEEE
DOI 10.1109/COMPSAC57700.2023.00022

during executions however these accesses must be coalesced to improve data locality and hence reduce the number of memory transactions required to serve a given computation.

Optimizing the performance of memory-intensive applications on GPUs thus requires an in-depth understanding of the memory hierarchies of the GPU architecture in order to maximize the data locality among threads. Developers need to understand how to coalesce memory accesses and how to manage control flow divergence of threads to leverage the benefits of the warp architecture and improve performance. Profiling applications give some insight into the data access patterns to improve data locality and aid in co-locating data and computations. This could also be leveraged to take advantage of the data persistence feature introduced by NVIDIA.

Typically, data that is frequently accessed should be protected from early eviction in order to reduce memory latencies and mitigate cache thrashing. Wang et al. [7] identify frequent access patterns based on user requests. Devashree et al. [8] through static profiling based on PTX code, obtain access patterns and analyze data reusability between thread blocks to determine data locality. Carlo et al. [9] and Fensch et.al [10] determine data access locality using complier-based hints and software generated self test approaches respectively.

In modern architectures, the data loads from the DRAM are cached in the L2 cache by default in an attempt to maximize memory bandwidth by using as much fast memory and as little slow-access memory as possible. However when executing many machine learning (ML) and high performance computation (HPC) applications, due to the relatively small ratio of cache sizes to input data sizes [11], there is the need to prioritize the data to be cached. Prior researchers [11] [12] [13] [14] [15] having observed that cache lines are sometimes evicted before they are accessed by the threads that need the data, have suggested different approaches to improve cache management.

With NVIDIA's L2 cache residency control feature, when a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be persisting. Data considered to be persistent are stored in a set-aside region and will be last in the eviction priority order. The persistence offered by this evict_last policy provides the opportunity to cache frequently accessed data and thus minimize the time spent in fetching newer cache lines from the global memory during executions. The residency control feature is particularly useful when the the data locality is high and the access patterns are highly coalesced per memory load.

We illustrate the implementation of L2 cache residency control for data requests from a single kernel and for concurrently running kernels when the data requests from the global memory by each kernel is greater than the available L2 cache memory.

Scenario 1: Single data request and residency control

With a hitRatio of 1.0 (Figure 1), a single kernel with access window twice the size of the set-aside area, will evict cache lines to keep the most recently used data from the access window, in the set-aside portion of the L2 cache because the

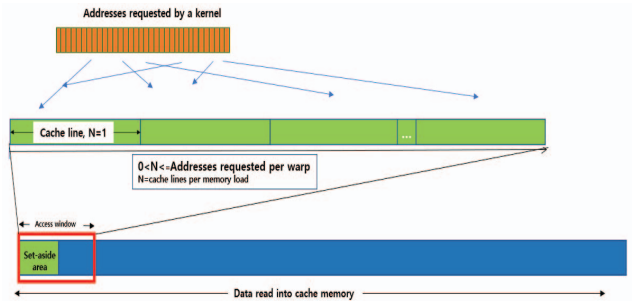

Fig. 1. Residency Control for a single kernel

set-aside area is smaller than the window. With uncoalesced access patterns this will further result in thrashing as the cache lines will not be maximized

Scenario 2: Concurrent data requests and residency control

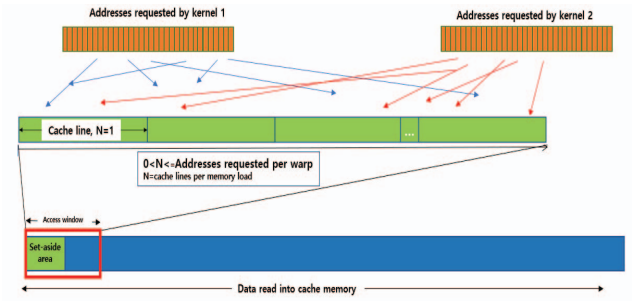

Fig. 2. Residency control for concurrent kernels

In the case of two concurrently running kernels from different applications(Figure 2), a hitRatio of 1.0 for both Apps will result in cache lines being evicted by each kernel to keep only the most recently used data each requires when competing for the shared L2 cache resource.

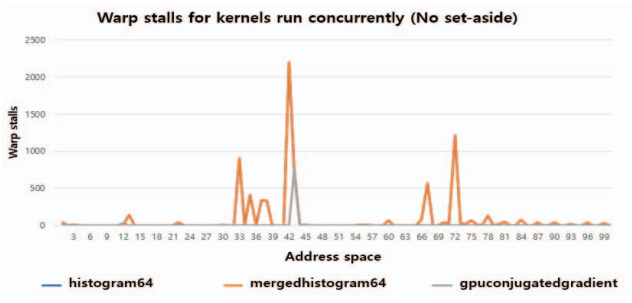

Fig. 3. No set-aside area in L2 cache [16]

Moreover, research by Adufu et al. [16] using the new NVIDIA residency control feature, revealed that when residency controls were enforced during the concurrent execution of two applications, warps were stalled for the histogram64 kernel (Figure 3 and Figure 4). This gives rise to the need

to characterize applications based on their access patterns in order to maximize the new L2 residency control feature.

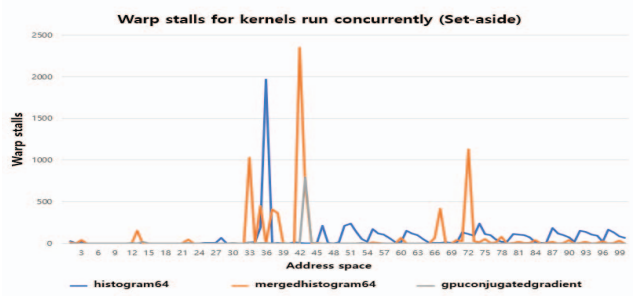

Fig. 4. Set-aside area in L2 cache [16]

From the above, it is imperative to quantitatively determine the amount of frequent accesses by the application and identify the access patterns in order to maximize the benefits of the L2 cache residency control feature.

III. Degree of Coalescing through Static Profiling

We adopt a static profiling approach based on the PTX code of a workload [17] [8]. Using this approach, we obtain the data accesses to addresses in global memory by threads in a warp.

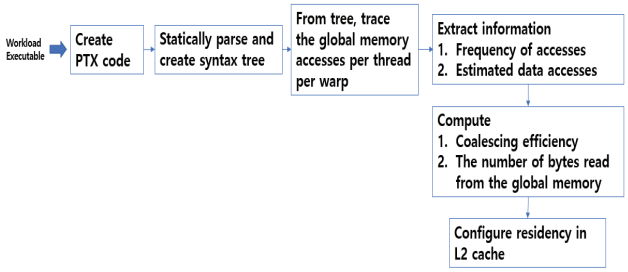

Fig. 5. Procedure for Static Profiling and Residency Control

From Figure 5, we first assemble a PTX code from the application's executable and then build a syntax tree using a modified PTX parser obtained from [18]. From the tree, we trace the global memory accesses by threads within each warp. This is derived based on addresses accessed using the **ld.global** command in the PTX code.

We then extract information on the frequency of accesses per data region and calculate a coalescing efficiency or the degree of coalescing using Equation 1 defined in Kim's dissertation [17] for a range of 128 Bytes per warp.

$$Degreeofcoalescing = \sum_{n=0}^{32} \frac{Coalescingsectors[n]}{warpcount} \quad (1)$$

The range of 128 Bytes is used for data locality analysis since it is synonymous to a cache line size of 128 Bytes. For each warp, the range is defined as 128 Bytes from the start position of the sector first accessed by a thread of the warp. We also assume that the size of the data read per access is constant at 4 Bytes per data read (1 data region) based on observations from the sizes of data accesses across some selected applications.

IV. L2 Cache Residency Configuration Based on Static Profiling

NVIDIA's L2 cache residency control feature [19], allows developers to influence the persistence of data in the L2 cache and hence lower latency accesses to global memory. However this is highly dependent on the frequency of data access by threads. Data accessed frequently and thus considered persistent are stored in a set-aside region. By so-doing, the data in the set-aside region is considered last in the eviction priority order.

```
cudaGetDeviceProperties(&prop, device_id);
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, prop.persistingL2CacheMaxSize);

cudaStreamAttrValue stream_attribute;
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr);
stream_attribute.accessPolicyWindow.num_bytes = num_bytes;  // Number of bytes for persisting accesses < set-aside area

stream_attribute.accessPolicyWindow.hitRatio  = 1.0;
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;

//Set the attributes to a CUDA stream of type cudaStream_t
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_attribute);
```

Fig. 6. L2 cache configurations [19]

Until now, configuring the access policy window (Figure 6) was dependent on recommendations by NVIDIA and guesswork. This approach lacks in quantitative analysis of the data access patterns and frequency from the global memory.

Another approach requires dynamically profiling the workload to obtain information of data transfers from the global memory to the threads through the caches. This however does not expose the access pattern which is necessary for configuring the streaming or persistence property in the access window.

From the static profiling approach described in Section 3, we obtain quantitatively, the frequency of data accesses to the global memory. With this extract, the configurations of the access window (Figure 6) can be leveraged to maximize residency control and performance for each workload.

The required configuration (Figure 6) for controlling the data residency in the L2 cache include the following: **1.** The size of the set-aside area **2.** The base data region for persistence **3.** The size of data reads to be considered for persistence or streaming policies. **4.** The hit ratio of accesses that receive the hitprop property. **5.** The hitprop property of either streaming, normal or persistence.

The size of the L2 cache set-aside area for persisting accesses may be adjusted within limits. NVIDIA recommends

that the maximum limit (75% of the L2 cache memory) be set-aside for persistence. Next is the base data region for the persistence. From our observations of running single applications on the NVIDIA A30 architecture, the data accesses begin from 1024 Bytes. The base pointer (ptr*) can thus be configured to point to the first data element that is copied from host to device using the cudaMemcpyAsync API.

The size of data reads to be considered for persistence in the L2 cache can be controlled using both the num_bytes and the hit-ratio. Our research work focuses on determining the size of the data reads to be considered in the access window (num_bytes) and determining the hitprop property to be applied to each application during execution.

A. L2 Access Window Size, num_bytes

When a warp executes a global memory access instruction, the efficiency of the global memory access is determined based on the memory addresses accessed by threads within the warp. From the profiled information, we are able to predict how many sectors will be accessed during the execution of each warp. We estimate the size of the accessWindowPolicy or num_bytes based on the estimated number of sectors (Estimated Sector Access, ESA) accessed per warp [17]. In GPU architectures, a sector has a size of 32 Bytes (B). Thus we compute the access window size (num_bytes) required to serve data requests per execution using Equation 2.

$$Num\_bytes = ESA \times 32 \times Warpcnt \tag{2}$$

B. Hit Property Through Access Pattern Characterization

When each thread in a warp accesses a contiguous data region within a cache line repeatedly, the number of cache lines required for a transaction reduces maximizing the benefits of data persistence. However, Adufu et al. [16] suggest that applying persistence to kernels with streaming data accesses may serve as a performance bottleneck. Thus, we considered the maximum data accesses per cache-line as well as the coalescing co-efficient of applications to characterize them as either streaming (S) or persistent (P) according to the caching policy given by NVIDIA.

We define a threshold of minimum accesses per cache line, α required for an application's accesses to be considered as persistent. We set the value of α to α **= 16,384** (50% * 1024 threads * 32 warps) based on the maximum number of accesses possible within 1 SM on the Ampere Architecture. For accesses greater than the defined threshold, α, we consider the coalescing efficiency to predict whether enforcing persistence would result in improved performance. We set a threshold for the coalescing co-efficiency as β, **0**$<\beta<$**1**, so that if an application's coalescing efficiency is higher than β = 0.5 then the application should be configured to implement L2 cache data persistence.

V. Experiment Environment

We evaluated the proposed approach by statically profiling some application workloads from Polybench [20] benchmarks in an environment described in Table I. We present the grid/block dimensions of the workloads used in our evaluation in Table II.

TABLE I
Experimental Set-up

GPU Device	NVIDIA A30
Device Memory	24GB
GPU memory bandwidth	933 GB/s
Cuda version	12.0
Nvidia-smi/ GPU Drivers	525.60.13
DCGM version	3.1.3
Nsight Compute version	2022.04

TABLE II
WORKLOAD GRID-BLOCK DIMENSIONS

WORKLOAD	GRID_X	GRID_Y	THREAD_X	THREAD_Y
2DConv_St	128	32	32	8
BICG	256	1	16	1
GEMM	2	8	32	8
Gramschmidt	8	1	256	1
2DConv_XLarge	512	32	32	1

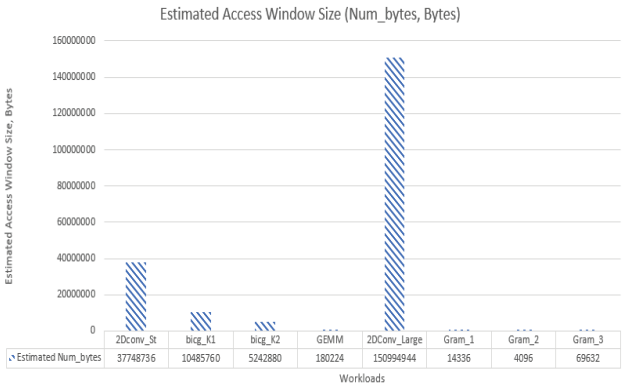

Fig. 7. Estimated Access Window Size per application

TABLE III
Access frequencies and coalescing efficiency per kernel

WORKLOAD	CE	Number of Warps	Frequent access	Access type
2DConv_St	0.92	294912	9216	S
BICG_K1	1	163840	1064960	P
BICG_K2	0.53	163840	1064960	P
GEMM	1	1408	384	S
Gram_1	1	448	6144	S
Gram_2	1	128	6144	S
Gram_3	1	2176	12288	S
2DConv_XLarge	0.92	1179648	18432	P

A. Observation 1: Frequency of accesses

From Table III, it was observed that the BICG kernels (BICG_K2 and BICG_K2) have the highest maximum number of accesses to the global memory, per cache-line. Applications

with large grid, block dimensions are more likely to benefit from the L2 cache residency controls than applications with small grid, block dimensions since the number of accesses increase with the number of threads concurrently accessing a particular data region. However, from Figure 7 BICG kernels had much lower estimated access window sizes compared to applications like 2DConv_XLarge. This meant that, though BICG had the greatest number of accesses, these accesses were to the same data regions within the cache-line.

B. Observation 2: Coalescing Graph of the Applications

From the coalescing graph (Figure 8) proposed by Kim [17], we can determine if frequent accesses are to the same data location or to different locations/sectors in the cache-line. When all threads access the same data location for instance, there is high coalescing and high data reuse as was in the case of BICG_K2. This however translates into smaller estimated access window sizes.

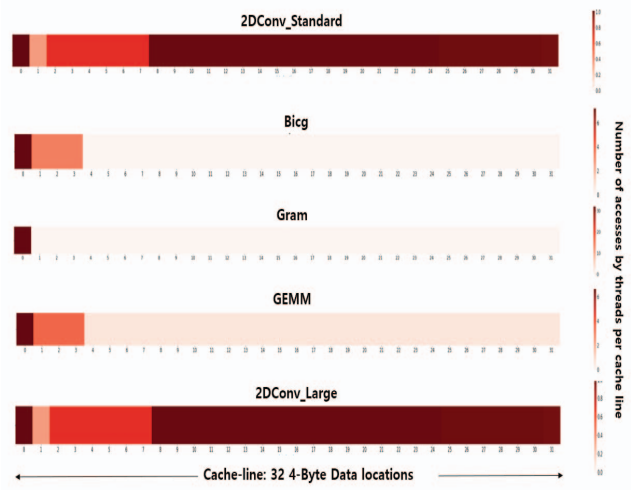

Fig. 8. Coalescing graph for workloads

We observed also that, the degree of coalescing was the same for 2DConvolution workloads, 2DConv_Standard and 2DConv_XLarge, regardless of the the data size or the grid/block dimensions of the workload. The 2DConv_Standard has smaller grid/block dimensions and a data set of (4096*4096) however it has the same coalescing efficiency as 2DConv_XLarge which has a bigger grid/block dimension and data set (16384 *16384).

Applications with large dimensions have the same average coalescing graph since the access patterns does not change with thread block dimensions but only increases the number of frequent accesses per data region. This is because the profiling of access patterns depend on how threads request data from global memory. Our observation makes it easier to estimate the total average number of accesses per data region.

C. Observation 3: Hit Property Through Access pattern Characterization

From Table III, we observed that kernels like GEMM, Gram_1, Gram_2, Gram_3 and 2DConv_Standard had less accesses than the defined threshold of α **= 16,384** thus we did not consider their coalescing efficiencies. For accesses greater than the defined threshold, α, we considered the coalescing efficiency before assigning the persistence hit property. For 2DConv_XLarge, BICG_K1 and BICG_K2 kernels, the CE values were higher than the threshold, β = 0.5 and thus were assigned the persistent access type, P.

D. Kernel performance using estimated access window sizes

We evaluated the performance of the 2DConv_Standard and 2DConv_XLarge workloads based on the estimated access window sizes as well as the values of characterization. We selected the 2DConv_Standard and 2DConv_XLarge workloads since they had the highest estimated access window sizes of 36MB and 144MB respectively. For our investigation, we adopted the recommendation of NVIDIA and set aside 75% of the L2 cache for persistence. This translated to a maximum of 18MB of L2 cache memory set-aside for persistent accesses. For each workload, we set the base pointer (ptr*) to the first data element that was copied from host to device using the cudaMemcpyAsync API and we set our hit-ratio to 1.0.

Applications with large dimensions have more threads and are thus more prone to data contention and warp stalls [16]. Thus using the Nsight Compute profiler [21], we validated our analysis by observing the warp stalls and L1 load throughput for the two workloads obtained for executions with and without the L2 cache configured. We configured the L2 caches based on the num_byte configurations represented in Figure 7.

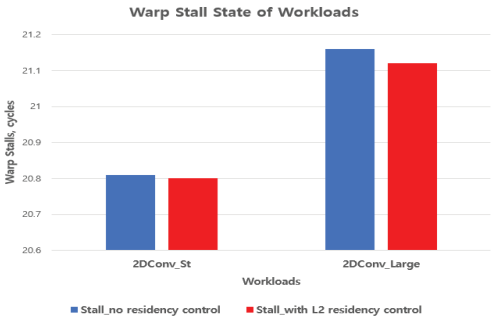

Fig. 9. Warp Stall State of Workloads

We observed that the stall long scoreboard remained relatively the same for the applications for both run-time scenarios revealing an accurate estimation of the L2 cache access window size. The slight improvements seen from 21.16 cycles for Stall_no residency control to 21.12 cycles for Stall_with L2 residency control during the execution of 2DConv_Large shows that for workloads with more frequent accesses, L2 helps reduce the warp stalls.

We also considered the L1 cache load throughput of three workloads to observe if there were improvements in the

load throughput. With our proposed approach, L1 cache load throughput increased by 2.75% for GEMM, 0.33% for 2DConv_St and 0.46% for 2DConv_Large respectively. This was as a result of increased data availability in the L2 cache during loads to the L1 cache.

VI. Related Works

Recent research works have focused on maximizing cache management. Duong et al. [22] simulate the use of reuse distance-based bypass policy with warp throttling to protect hot cache lines from early eviction. Fang et. al [23] deploy FIFO buffers to reorder memory requests and as a result shorten the reuse distance of memory requests before they are sent to L1 caches. However, these works focus on maximising the L1 cache and not the L2 cache.

Static profiling analysis based on PTX code [8] [13] [17] has been used to expose data reusability between thread blocks as well as the data locality that exists between mutual thread blocks. However, this does not reflect the global memory access frequencies nor the size of data expected to be fetched into the L2 cache memory.

Walden et al [6] make use of the new L2 residency control feature as a memory optimization technique for a sparse linear algebra kernel. Even though from their experiments, the use of L2 persistence and asynchronous memory copies improve the overall performance by 81.2%, they mentioned that they were unable to explore the impact of the configured L2 set-aside area on performance.

Conclusion and Future Work

This paper analyzed L2 cache access patterns for selected workloads using static profiling of PTX code. Through a trace of syntax trees built for each workload, we determined the degree of coalescing for threads within a warp, the frequency of data access, and estimated the size of data region that would be considered for L2 residency controls as the access window. Additionally, we classified the workloads based on the frequency of accesses and the degree of coalescing.

We validated our approach by comparing the configurations from our approach for two workloads with different number of frequent accesses. Our proposed approach did not result in any further warp stalls like those observed by Adufu et. al [16]. With our proposed approach, we also observed L1 cache load throughput increases by 2.75% for GEMM, 0.33% for 2DConv_St and 0.46% for 2DConv_Large respectively as data is more resident in cache.

In the future, we intend to explore the use of static profiling to determine the L2 cache persistence size during scheduling of different workloads.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2021R1A2C1003379).

References

[1] IBM Cloud Server's NVIDIA GPU, https://www.ibm.com/kr-ko/cloud/gpu?mhsrc=ibmsearch_a&mhq=GPU, (Last accessed, January 27, 2023)
[2] Recommended GPU instances, https://docs.aws.amazon.com/ko_kr/dlami/latest/devguide/gpu.html, (Last accessed, January 27, 2023)
[3] Cloud GPU, https://cloud.google.com/gpu?hl=ko, (Last accessed, January 27, 2023)
[4] Elastic GPU, https://www.alibabacloud.com/ko/product/gpu, (Last accessed, January 27, 2023)
[5] NVIDIA A100 datasheet, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf,
[6] A. Walden, M. Zubair, C. P. Stone and E. J. Nielsen, "Memory Optimizations for Sparse Linear Algebra on GPU Hardware," 2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), 2021, pp. 25-32, doi: 10.1109/MCHPC54807.2021.00010.
[7] Danqi Wang; Chai Kiat Yeo, "Exploring Locality of Reference in P2P VoD Systems," Multimedia, IEEE Transactions on, vol.14, no.4, pp.1309, 1323, Aug.2012
[8] D. Tripathy, A. Abdolrashidi, Q. Fan, D. Wong, and M. Satpathy, "Localityguru: A ptx analyzer for extracting thread block-level locality in gpgpus," in 2021 IEEE International Conference on Networking, Architecture and Storage (NAS), pp. 1–8, IEEE, 2021.
[9] Di Carlo, S.; Prinetto, P.; Savino, A., "Software-Based Self-Test of SetAssociativeCache Memories," Computers, IEEE Transactions on, vol.60, no.7, pp.1030,1044, July 2011 doi:10.1109/TC.2010.166
[10] Fensch, C.; Barrow-Williams, N.; Mullins, R.D.; Moore, S., "Designing a PhysicalLocality Aware Coherence Protocol for Chip Multiprocessors," Computers, IEEE Transactions on , vol.62, no.5, pp.914,928, May 2013
[11] Lal, S., Sharat Chandra Varma, B., and Juurlink, B. (2022). A Quantitative Study of Locality in GPU Caches for MemoryDivergent Workloads. International Journal of Parallel Programming, 50, 189-216. https://doi.org/10.1007/s10766022007292
[12] Chen X., Chang L., Rodrigues C., Ly Jie., Wang Z. and Hwu W., (2014) Adaptive Cache Management for Energy-Efficient GPU Computing, MICRO-47: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture December 2014697 pages ISBN: 9781479969982
[13] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," ACM Transactions on Architecture and Code Optimization (TACO), vol. 18, no. 3, pp. 1–26, 2021.
[14] Rogers, T.G., O'Connor, M.and Aamodt, T.M., Cache-conscious wavefront scheduling. In: Proceedings of the 45th annual IEEE/ACM international symposium on microarchitecture, MICRO (2012)
[15] Jia, W., Shaw, K.A., Martonosi, M.: Characterizing and improving the use of demand-fetched caches in GPUs. In: Proceedings of the 26th ACM international conference on supercomputing, ICS (2012)
[16] Adufu, T. and Kim Y. (2022). A Performance Benchmark of Cached Data Access Patterns on GPUs. KNOM Review '22-02 Vol.25 No.02, pg 30-39. https://doi.org/10.22670/knom.2022.25.2.
[17] Kim, Jieun (2023). A Study on Data Locality and L1 Cache Analysis of GPU Workload Using Static Profiling, Masters' Thesis, Seoul National University, Seoul
[18] https://github.com/JIeunAmy/coalescing_graph_with_PTX
[19] Sliding Window Experiment, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations
[20] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. http://www. cs. ucla. edu/pouchet/software/polybench. (Last accessed, February 17, 2023)
[21] Nsight Compute, https://developer.nvidia.com/nsight-compute
[22] Duong, Nam, et al. "Improving Cache Management Policies Using Dynamic Reuse Distances." 45th Annual IEEE/ACM International Symposium on Microarchitecture, Pages 389-400, 2012
[23] Fang Juan, Zelin Wei, and Huijing Yang. 2021. "Locality-Based Cache Management and Warp Scheduling for Reducing Cache Contention in GPU" Micromachines 12, no. 10: 1262. https://doi.org/10.3390/mi12101262