Optimizing Performance Using GPU Cache Data Residency Based on Application's Access Patterns

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*—Memory management is a significant aspect of executing applications on GPUs even in the cloud environment. With the advancements in GPU architecture, issues such as data reuse, cache line eviction and data residency are to be considered when optimal performance for concurrently running applications. Frequency of data access from global memory has significant impact on the performance of the application with increased latencies when accesses result in cache misses. Through static profiling, we identify the access patterns to the global memory and investigate the relationship between frequent access patterns and data residency in the cache. From our investigations, we observed that each application frequently accesses a data region in memory though the range of addresses accessed differ. We evaluated our estimated set-aside area for LSTM and CSR applications. Executions using our proposed estimations shows a speed-up in the performance LSTM (1.004x) while CSR experienced a slow-down (0.998x) when both were co-executed with their respective estimated set-aside areas.**

***Index Terms*—Static Profiling, Frequently Accessed Data, Data Residency**

I. Introduction

Graphics Processing Units (GPU) provide high computational capacity for compute intensive applications such as High Performance Computing (HPC) applications. However, there remains a bottleneck in performance mostly as a result of misaligned memory accesses leading to high miss rates. To mitigate this phenomenon, researchers [1] [2] [3] have proposed the use of different approaches including the use of FIFO buffers [4] for instance, to reorder memory requests and as a result shorten the reuse distance of memory requests before they are sent to L1 caches. Additionally, GPUs can hide memory access latencies with computation as multiple threads execute the same instruction in parallel however the access patterns of applications are hardly leveraged to improve performance.

NVIDIA, with it's Ampere architecture for instance, offer a new feature that allows the user to leverage data persistence in a defined portion of the L2 cache for applications with high frequent accesses [5]. This is an effort to reduce early eviction of data thus ensuring that data that is frequently accessed is available during the execution lifetime of the application hence lowering access latencies.

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

On the other hand, Recurrent Neural Network (RNN) like Long Short-Term Memory (LSTM) use recurrent weights between GEMM operations, and thus require frequent accesses to global memory. The recurrent weights in these networks can be made persistent in L2 and re-used between GEMM operations [5].

This paper proposes the use of static profiling of PTX code to determine the access frequencies of data read from the global memory. By a static profile analysis of the application, a memory access profile is created to show the memory region that is accessed through out the application's execution lifecycle. From this, regions that are frequently accessed can be identified and applications can be classified into either streaming, normal or persistent categories based on the access frequencies to the data regions. The range of frequently accessed memory addresses can then be marked for persistent data storage.

Our research provides a basic approach for application classification to leverage the benefits of L2 data residency on modern GPU architectures. Through this study,

- We determined the data access frequencies of applications through static profiling and create data access profiles for the applications
- We classified the applications into three groups; persistent, normal and streaming based on a frequency score
- We investigated the size of persistent area required for optimal performance for an application
- We evaluated the performance of selected applications when co-scheduled with different applications

The rest of the paper is organized as follows: in Section 2, we briefly describe the background and motivation for this study and give details of the proposed static profiling approach in Section 3. In Section 4, we present the results of our study and highlight some related works in Section 5. We conclude the paper in Section 6.

II. BACKGROUND AND MOTIVATION

When threads in a warp access a contiguous data region from the cache repeatedly, the number of cache lines required for transactions from the global memory reduces thus maximizing the benefits of newly introduced features such as data persistence or data residency. However, if such accesses are not contiguous though repeated, the cache memory fills up quickly when the data requested from global memory is very large. This exposes previously loaded data to early eviction and warp stalls and eventually leads to higher latency during execution.

We investigate the effect of data availability in the cache on time and warp stalls, for an LSTM application [6] using the Nsight Compute profiler [7]. We profile the LSTM application for two cache control scenarios: flush all and flush none. The flush all option for cache control in Nsight Compute clears the cache of data previously loaded during profiling whilst the flush none keeps the data in cache during profiling.

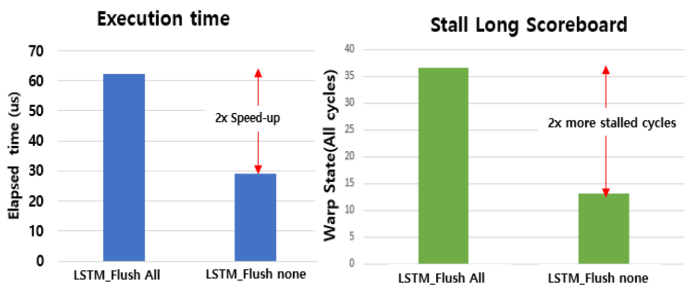

Fig. 1. LSTM behaviour in two cache control scenarios

Figure 1 shows a 2x speed-up in the execution time of the LSTM kernel when data loads are allowed to persist in the cache memory using the **flush none** option for the cache controls in Nsight compute profiler. This also translates in significant reduction (2x) in the warp stalls observed during the execution of the kernel.

From these experiments, we observed that applying data residency to applications such as LSTM could result in significant improvements in latency and may also mitigate over-all performance slow-downs for concurrently running applications scenarios. Following this observation, we investigated the relationship between frequent access patterns and data residency in the cache using a static data access profiling approach.

III. Data Access Profiles Based on Static Profiling

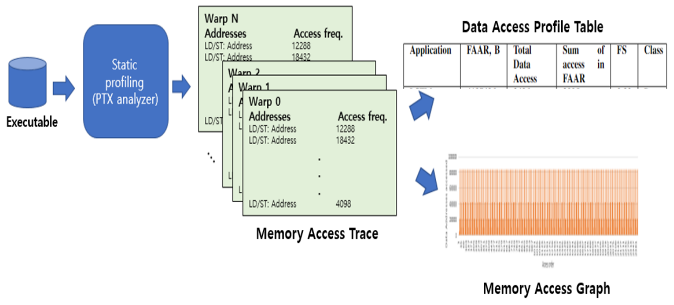

Fig. 2. Data Access Profiling Process

Figure 2 describes a method of obtaining the data access patterns of the application through static profiling from the global memory. We begin the process by assembling a PTX code from the application's executable and use a modified PTX parser obtained from [8] to obtain information for the data access profile. Using the parser, the thread-to-memory relationship in terms of thread ID, block ID, memory addresses accessed and other kernel parameters are obtained using the **ld.global** command. This information can be used to capture inter-thread, inter-warp, inter-TB locality within the same kernel as well as across multiple kernels [9] .

A. *Data Access Granularity*

Blocks are divided into warps of 32 threads with every thread in the warp executing the same instruction in lock-step manner but on different data. When a warp executes an instruction that accesses memory, the requests are processed together for all the threads within the warp. Thus we extract the access frequencies at the warp granularity.

We obtain the number of memory access by threads in a warp for a given address range synonymous to a cache line. An address range of 128 Bytes per cache is selected since it is synonymous to a contiguous data region for a cache line of size 128 Bytes in the Ampere architecture we use for our experiments. For each data region, the range is defined as 128 Bytes from the start position of the sector first accessed by a thread within the warp [8]. We record the access frequency per cache line accessed by each warp.

B. *Data Access Profile*

Using the information obtained, we create a data access profile for each application. The data access profile is expressed both graphically and in tabular form (Figure 2). The data access graph shows only the application's access patterns and frequencies. However, the data access profile table contains additional information derived from further analysis. This includes the Frequently Accessed Address Range **(FAAR)**, the Sum of regions within FAAR, the total memory regions, the Frequency Score **(FS)** and the class of the application.

The Frequently Accessed Address Range **(FAAR)**, is the memory address region frequently accessed by the application during the application's life-cycle, in bytes. This can be seen as the dense parts of the data access graph. The number of accessed regions within this repeatedly accessed data region during the execution of the application is known as the **Sum of Accesses in FAAR**. This metric is particularly useful in determining the class of the application as well as the **Total Data Accesses** accessed by the application.

C. *Application classification Approach*

For quantitative analysis and application classification, we calculate a Frequency Score (FS), which is the ratio of all memory regions accessed in the Frequently Accessed Address Range (FAAR) to the Total Data Accesses by the application as shown in equation 1.

$$FrequencyScore, FS = \frac{\sum AccessinFAAR}{TotalDataAccess} \tag{1}$$

Based on the Frequency Score (high or low), applications can be classified into one of three classes: streaming(S),

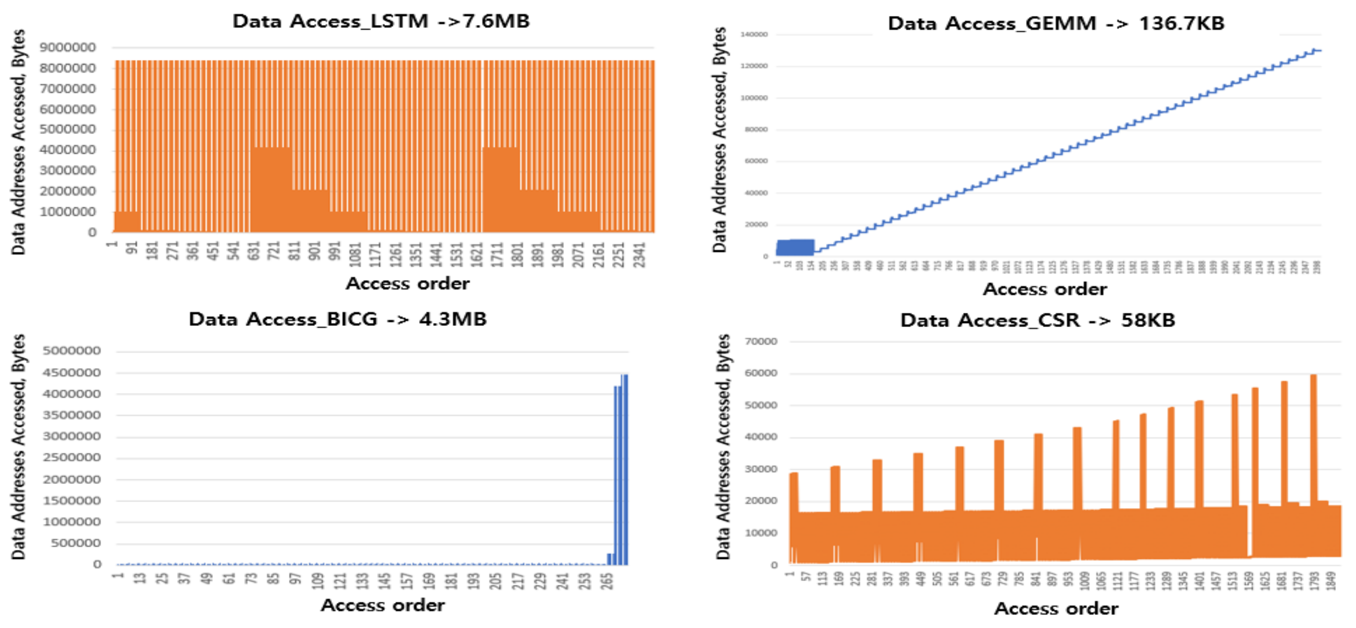

Fig. 3. Data access profiles for selected applications

normal (N) and persistent (P). Our classification is highly dependent on the frequency of data access throughout the application's execution life-cycle which serves as a reliable measure. We acknowledge that, there could be multiple contiguous accesses by different threads to the same memory address which could affect the distribution of accesses to the addresses regions. With our static profiling approach, we consider each entry and exit of threads to a given data region in the global memory as an access order and do not consider multiple contiguous loads from the same data region separately.

IV. Experiments and Results

A. *Application Classification and Persistent Area*

We statically profile four applications: Long Short Time Memory (LSTM), Compressed Sparse Row (CSR), General Matrix Multiplication (GEMM) and BiConjugate Gradients (BICG) from the Tango Benchmarks [6], [10] and Polybenchmark [11] respectively on one (1) NVIDIA A30 (Ampere architecture) GPU. We compiled each application with CUDA version 12.0 before generating the PTX code. Table I shows the grid/block dimensions of the workloads used during the static profiling analysis.

TABLE I
APPLICATION GRID-BLOCK DIMENSIONS

WORKLOAD	GRID_X	GRID_Y	THREAD_X	THREAD_Y
LSTM [6]	1	1	100	1
CSR [10]	8	1	512	1
GEMM [11]	2	8	32	8
BICG [11]	256	16	16	8

Since we do not consider multiple contiguous memory requests to the same memory address separately in this research, we assume that the accesses to memory follows a normal distribution throughout the application's execution life-cycle. We define three classes according to NVIDIA's caching policies [12] and apportion an FS score range to each class. For a score within the range, 0<FS<0.33, the application is classified as Streaming(S). For a score within the range 0.33<FS<0.66, the application is classified as Normal(N). When data is accessed frequently giving an FS score within the range 0.66<FS<1, the application is classified as Persistent(P).

TABLE II
DATA ACCESS PROFILE TABLE

Application	FAAR, B	Total Data Access	Sum of access in FAAR	FS	Class
LSTM [6]	4197496	2406	2005	0.83	P
CSR [10]	20000	1882	1795	0.95	P
GEMM [11]	10364	2416	292	0.12	S
BICG [11]	51324	276	264	0.96	P

Observation 1: Frequently Accessed Address Region (Persistent Area)

From Table II we observed that, all applications frequently accessed data within a given range. The range however varied for each application. LSTM application for instance had a uniformly repeated access pattern to data regions up to 4197496B (4MB) though it accessed data over a 7.6 MB range. GEMM on the other hand accessed a range of 10364B (10KB) repeatedly at the beginning of the execution and later streamed data from different memory locations up to 127KB.

The identified range for repeated accesses over an application's execution life-cycle can serve as the size of memory reserved for persistence.

Observation 2: Frequency Score (FS) and Classification of Applications

The Frequency Score (FS) for each application was calculated relative to the range of data regions accessed by the application. From Table II, we observed that BICG application had the highest FS of 0.96. This shows that most of the data it accessed was within the range identified as the frequently accessed region. It was therefore classified under applications with persistent accesses. CSR and LSTM were next with FS of 0.95 and 0.83 respectively which corresponded to the access patterns depicted in the graph. GEMM on the other hand could be classified as a streaming application as shown in Figure 3 and from the FS score of 0.12 in Table II.

Observation 3: Data Access Characteristics

We also observed from Table II that, applications running on the GPU frequently accessed data within a relatively small range compared to the size of the L2 cache memory. For instance, the sizes of the frequently accessed memory regions for CSR (19KB) and BICG (50KB) were very small compared to that of LSTM (4MB). The range of persistent accesses varied for each application suggesting that even though some applications may have high FS, they may not require the L2 cache residency feature.

B. Evaluating Performance For Different Sizes of Persistent Area

In this research, we evaluated the residency control feature for concurrently running applications as well. Compared to other applications, LSTM had the widest range of repeated data accesses which can affect cache performance. Thus, we investigated the effect of allocating different sizes of persistent area to the LSTM application when executed alone.

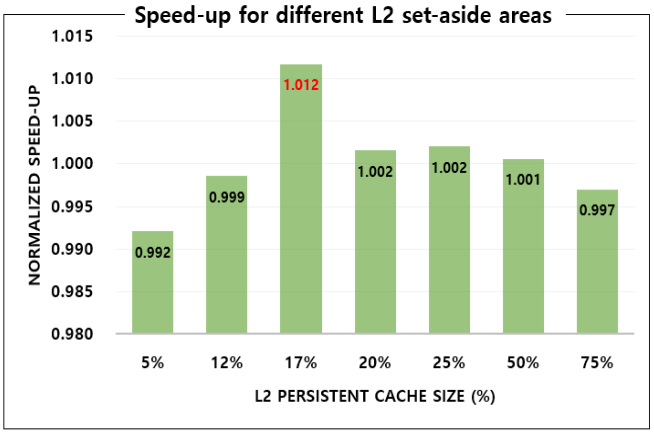

Fig. 4. Performance for various allocations of L2 Persistent area

From Figure 4 we observed that, while the LSTM application was being executed alone, allocating different sizes of the L2 cache set-aside area affected the performance of the application differently. For instance, when 12% of the cache memory was set-aside, it resulted in the worst performance. We attributed this to the fact that not all the data required during the execution process was kept in the set-aside area. In such a case, increased cache misses to the set-aside area would require additional accesses to the global memory leading to higher latencies.

We also observed that, by assigning a set-aside area of 17%(4MB) using the estimation obtained from our proposed approach, LSTM application experienced the best performance with the highest speed-up. This re-emphasized the need to accurately estimate the set-aside area of the L2 cache based on the application's characteristics. We assumed that A30 allows L2 cache to be set-aside for persistent accesses in 1/16th increments (1.5 MB) as it is for the A100 [5]. Consequently, by comparing the performance of the estimated set-aside area to other sizes (+/- 1.5MB), we observed that allocating additional set-aside area than required by the application may result in slowdowns instead of the intended speedup.

C. Evaluating Performance For Concurrent Executions in Different Scheduling Scenarios

We investigated the performance of residency controls when executing two applications concurrently for different scenarios.

Scenario 1: Set-Aside (SA) for LSTM only

We investigated the effect of data residency on performance when co-running LSTM with CSR application which does not have any persistent area allocated to it. We compared the results with the performance of LSTM when running alone (No Set-Aside, NSA) in Figure 5.

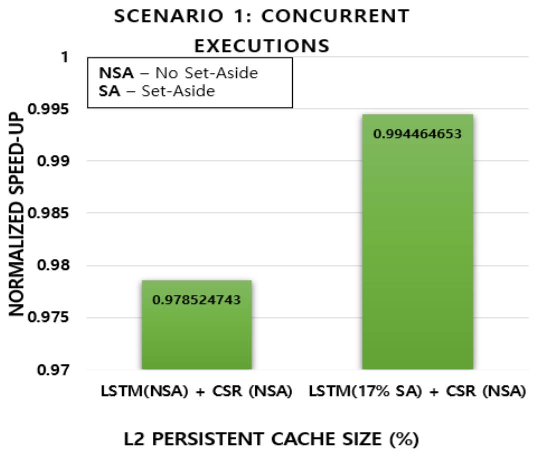

Fig. 5. Performance for concurrent executions

From Figure 5 we compared co-executing LSTM (17% SA) with CSR (NSA) to co-executing LSTM (NSA) with CSR (NSA) and compare the results. From the experiments, we observed that when LSTM was co-executed with CRS, there was a degradation in performance. This however improved when a set-aside area of 17% was allocated to LSTM. We allude the improved performance to the allocation of the set-aside area in LSTM which facilitates faster access to data in the L2 cache.

Scenario 2: Set-Aside (SA) for both LSTM and CSR

We also investigated the effect of set-aside areas on performance when both co-running applications have different set-aside areas allocated to them. We compared the results for set-aside values allocated using our proposed estimations and otherwise to the performance of each application when run alone. The results are presented in Figure 6.

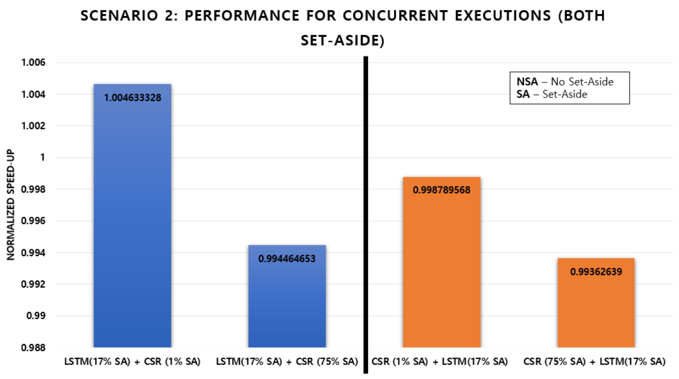

Fig. 6. Performance for concurrent executions(Both set-aside)

From Figure 6 it was observed that, when co-executed with set-aside areas allocated according to our estimations, while LSTM experienced a speed-up of 1.004x, CSR experienced a slow-down of 0.998x when both were co-executed with their respective estimated set-aside areas. We observed also that when 75% of the set-aside area is allocated to CSR, and 17% is allocated to LSTM, LSTM(17% SA) experiences performance degradation as the performance was 0.994x compared to the performance when LSTM (17% SA) was executed alone. On the other hand, CSR (0.993x) also experiences performance degradation when 75% of the set-aside is allocated to it and run concurrently with LSTM(17% SA). This suggests that, when the total allocation for persistent area for all concurrently running applications exceed the maximum 75% set-aside, there would be performance degradation of all the applications concurrently running on the GPU.

We also observed that, applications whose frequent data access range is less than 1.5MB such as CSR experienced were more prone to performance degradation thus, we do not recommend the use of the L2 cache residency control feature for such applications.

V. Related works

The design of modern GPU architectures reveal an attempt to maximize memory bandwidth by using as much fast memory and as little slow-access memory as possible hence improving over-all performance. Prior research works [9], [13]–[15] have attempted to identify access patterns and analyze data reusability between thread blocks to maximize the gains from data locality among threads. Also according to Walden et al. [2], the data layout of applications influence the effective utilization of memory bandwidth in GPU architectures.

Research works [3], [16]–[19], having discovered 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. Some works focused on determining the reuse of data stored in memory, seek to protect frequently accessed data from early eviction and hence improve performance. However, these works do not classify applications based on the frequency of data access.

In order to maximize the benefits of new features introduced in modern GPU architectures such as the L2 cache residency control feature, it is imperative to quantitatively determine the amount of frequent accesses by the application and identify the access patterns to the global memory. Degioanni's StAMP [20], propose a memory access profile which can be used by off-line scheduling strategies to minimize interference overhead. However, they did not consider the frequency of access to data regions.

In our research, creating a data access profile serves as a basis for classifying an application. From the data access profiles, data regions with continuous access frequencies can be identified and explored to influence data residency decisions.

Conclusion and Future Work

This paper implements a static profiling analysis to identify the access patterns of selected applications to global memory when executed on NVIDIA's A30 GPU. From our investigations, we observed that each application accesses a given memory region repeatedly. We classify the applications into three groups based on the frequency of access throughout the life-cycle of the application. We also estimated the optimum set-aside area for the LSTM application and evaluated the performance of the application for various co-scheduling scenarios. We ascertained that accurately estimating the set-aside areas is directly correlated to the performance of the application applying the data residency feature.

Acknowledgment

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

References

[1] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, BC, Canada, 2012, pp. 389-400, doi: 10.1109/MICRO.2012.43.

[2] 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), St. Louis, MO, USA, 2021, pp. 25-32, doi: 10.1109/MCHPC54807.2021.00010.

[3] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. 2021. PAVER: Locality Graph-Based Thread Block Scheduling for GPUs. ACM Trans. Archit. Code Optim. 18, 3, Article 32 (September 2021), 26 pages. https://doi.org/10.1145/3451164

[4] Fang J, Wei Z, Yang H. Locality-Based Cache Management and Warp Scheduling for Reducing Cache Contention in GPU. Micromachines. 2021; 12(10):1262. https://doi.org/10.3390/mi12101262

[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] Karki, Aajna Keshava, Chethan Shivakumar, Spoorthi Skow, Joshua Hegde, Goutam Jeon, Hyeran. (2019). Tango: A Deep Neural Network Benchmark Suite for Various Accelerators. 137-138. 10.1109/ISPASS.2019.00021.

[7] Nsight Compute, https://developer.nvidia.com/nsight-compute.

[8] GitHub Repository: JIeunAmy/coalescing_graph_with_PTX, URL: https://github.com/JIeunAmy/coalescing_graph_with_PTX.

[9] D. Tripathy, A. Abdolrashidi, Q. Fan, D. Wong and M. Satpathy, ”LocalityGuru: A PTX Analyzer for Extracting Thread Block-level Locality in GPGPUs,” 2021 IEEE International Conference on Networking, Architecture and Storage (NAS), Riverside, CA, USA, 2021, pp. 1-8, doi: 10.1109/NAS51552.2021.9605411.

[10] Programming Massively Parallel Processors Source Code, URL: https://github.com/junstar92/parallel_programming_study/blob/master-/CUDA/sparseMatrixVectorMul/SpMV.cu.

[11] Pouchet, Louis-Noël. ”Polybench: The polyhedral benchmark suite.” URL: http://www.cs.ucla.edu/pouchet/software/polybench.

[12] Kernel Profiling Guide, https://docs.nvidia.com/nsight-compute/ProfilingGuide/.

[13] D. Wang and C. K. Yeo, ”Exploring Locality of Reference in P2P VoD Systems,” in IEEE Transactions on Multimedia, vol. 14, no. 4, pp. 1309-1323, Aug. 2012, doi: 10.1109/TMM.2012.2191942.

[14] S. Di Carlo, P. Prinetto and A. Savino, ”Software-Based Self-Test of Set-Associative Cache Memories,” in IEEE Transactions on Computers, vol. 60, no. 7, pp. 1030-1044, July 2011, doi: 10.1109/TC.2010.166.

[15] C. Fensch, N. Barrow-Williams, R. D. Mullins and S. Moore, ”Designing a Physical Locality Aware Coherence Protocol for Chip-Multiprocessors,” in IEEE Transactions on Computers, vol. 62, no. 5, pp. 914-928, May 2013, doi: 10.1109/TC.2012.52.

[16] X. Chen, L. -W. Chang, C. I. Rodrigues, J. Lv, Z. Wang and W. -M. Hwu, ”Adaptive Cache Management for Energy-Efficient GPU Computing,” 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014, pp. 343-355, doi: 10.1109/MICRO.2014.11.

[17] T. G. Rogers, M. O’Connor and T. M. Aamodt, ”Cache-Conscious Wavefront Scheduling,” 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, BC, Canada, 2012, pp. 72-83, doi: 10.1109/MICRO.2012.16.

[18] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In Proceedings of the 26th ACM international conference on Supercomputing (ICS ’12). Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/2304576.2304582

[19] Lal, S., Varma, B.S. and Juurlink, B. A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads. Int J Parallel Prog 50, 189–216 (2022). https://doi.org/10.1007/s10766-022-00729-2

[20] Degioanni, Théo, Puaut, Isabelle. ”StAMP: Static Analysis of Memory access Profiles for real-time tasks.” In WCET 2022 - 20th International Workshop on Worst-Case Execution Time Analysis, Jul 2022, Modena, Italy. ⟨10.4230/OASIcs.WCET.2022.1⟩.