KNOM Review '22-02 Vol.25 No.02
https://doi.org/10.22670/knom.2022.25.2.30

GPU 데이터 캐시 접근 패턴 방법에 따른 성능 변화 분석

테오도라 아두푸◆, 김 윤 희°

A Performance Benchmark of Cached Data Access Patterns on GPUs

Theodora Adufu◆, Yoonhee Kim°

요 약

GPU는 다른 도메인의 응용 프로그램에 훨씬 더 높은 명령 처리량과 메모리 대역폭을 제공하므로 최근 범용 응용 프로그램을 성공적으로 가속화했다. 그러나 큰 메모리 대기 시간으로 인해 GPU 성능에 병목 현상이 남아 있습니다. 캐시는 칩 외부 메모리 트래픽을 줄이지 만 캐시 관리는 어렵다. Nvidia Ampere 아키텍처에 도입된 새로운 상주 제어 기능을 통해 사용자는 이제 캐시에 상주하는 데이터의 양을 제어할 수 있다. 그러나 여러 애플리케이션을 동시에 실행하는 동안 주의할 점은 데이터 지속성이 필요한 애플리케이션과 그 양을 식별하는 것이다. 이 백서에서는 처리량 및 데이터 액세스 빈도로 워크로드를 특성화하고 세 가지 공동 스케줄링 시나리오를 실험하여 최적의 성능을 위한 영구 캐시 할당을 결정한다. 서로 다른 데이터 액세스가 있는 응용 프로그램을 함께 실행할 때 L2 별도 할당이 지속적인 데이터 액세스가 있는 응용 프로그램에 편향되어서는 안 된다는 것을 관찰했다.

Key Words : L2 cache, Memory Access Pattern, Set-aside aware, GPU, L2 Access Cache Window Policy

ABSTRACT

GPUs have successfully accelerated general-purpose applications in recent times as they provide much higher instruction throughput and memory bandwidth to applications from different domains. There remains however a bottleneck in the performance of GPUs due to large memory latencies. Caches reduce off-chip memory traffic however, managing caches is difficult. With the new residency control feature introduced in the Nvidia Ampere architectures, users can now control how much data is resident in the cache. During co-executions of multiple applications, the caveat however is to identify which application requires data persistence and by how much. In this paper, we characterize workloads by throughput and data access frequencies and experiment with three co-scheduling scenarios to determine persistent cache allocations for optimum performance. We observed that when co-executing applications with different data accesses, L2 set-aside allocations should not be biased towards applications with persistent data accesses.

※이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2021R1A2C1003379).

◆ First Author : Sookmyung Women's University Department of Computer Science, theoadufu@sookmyung.ac.kr
° Corresponding Author: Sookmyung Women's University Department of Computer Science, yulan@sookmyung.ac.kr

논문번호 : KNOM2022-02-04, Received December 1, 2022; Revised December 10, 2022; Accepted December 16, 2022

Ⅰ. Introduction

Graphics Processing Units (GPUs) in comparison to traditional CPUs, provide much higher instruction throughput and memory bandwidth during the execution of applications [1]. Since the introduction of NVIDIA's Compute Unified Device Architecture (CUDA), many applications including High Performance Computing (HPC) scientific applications [2][3][4][5][6][7][8][9], have leveraged the higher capabilities to run faster on the GPU. As researchers and private data centers have increased their deployments of applications on GPUs, shared clouds like Microsoft Azure[10], Amazon EC2[11] also provide GPU-based infrastructure services to support GPU clouds. There remains however, a bottleneck in the performance of the GPU. GPUs can hide memory access latencies with computation however, misaligned memory accesses, poor data locality in the cache memory, cache thrashing, high miss rates and poor thread and block size configurations can have an expensive impact on performance. HPC applications for instance, seek to exploit more parallelism through the use of multiple threads, however these active threads contend for limited GPU cache resources during execution. This results in cache thrashing and high miss rates. Memory optimization is thus, the most important area for performance improvement. This paper investigates the use of L2 cache residency control as a means of optimizing memory and thus improving performance.

● We characterize applications by their data request sizes and by the frequency of data reuse.
● We apply residency control to optimize memory and improve performance.
● We investigate the impact of the size of the L2 set-aside cache area on the performance of applications when run concurrently.

In summary, this is a quantitative study on exploiting data access frequency in the L2 cache for a set-aside-aware execution of applications which improves performance.

The paper is organized as follows: in Section 2, we briefly discuss some related works on optimizing L2 cache performance. We give a background of the heterogeneous memory system of the GPU architecture and the L2 Cache residency control feature in Section 3. We explain our experimental setup in Section 4 and present the quantitative results in Section 5. We conclude the paper in Section 6.

Ⅱ. Related Works

Multiple memory optimization techniques and approaches have been employed to mitigate the effect of memory limitations. Though there has been several studies to exploit the data locality in GPUs [12] [13], Sohan Lal et al.[14] argue that there is a lack of quantitative analysis of data locality in GPUs.

2.1. Thrashing Improvement Techniques

When applied to GPUs, cache bypassing proposed in CPUs as a thrashing-resistant technique against early eviction in cachelines may not achieve the expected improvement[15]. Cache-conscious wavefront scheduling (CCWS) [13] improves the L1 cache hit rate in GPUs by alleviating inter-warp contention. These techniques however do not consider the size of reused data to improve performance.

2.2. Sectored Caches

Recent GPU architectures including the Nvidia Ampere architecture employ the use of sectored caches to fetch only the sectors that are requested instead of always fetching all the sectors of a cache line. In A30, a sectored-cache has a cache line size of 128 bytes (B), divided into four sectors. On a miss, a sector-cache will only fetch the 32 B sectors that are requested. A full cache

line is not automatically fetched however if all four sectors are requested, it is possible to fetch a full cache line. This leads to improved bandwidth utilization even for strided acceses. Jia. et al [12] explore the use of sectored caches to tackle issues such as data over-fetch and hence improve performance.

2.3. Residency Control

Walden et al [16] in an effort to maximize memory bandwidth utilization for a sparse linear algebra kernel explore the L2 residency control and the asynchronous copy instruction features of the A100 architecture. From their experiments, the use of L2 persistence and asynchronous memory copies improve the overall performance by 81.2%, which is slightly better than the original mapping algorithm with the new A100 features. They however did not explore the impact of the size of the L2 set-aside area on performance.

Ⅲ. GPU Memory Architecture

The GPU contains multiple small hardware units called Streaming Multiprocessors (SMs), on-chip L2 cache and a high bandwidth DRAM also known as the global memory (Fig. 1).

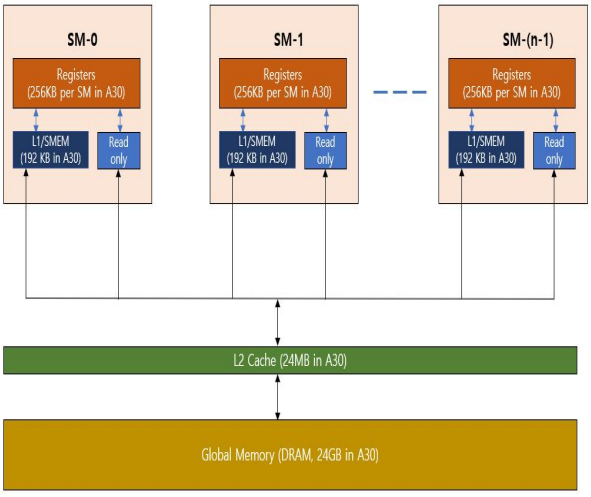

Fig. 1. Memory architecture of A30

The SMs can execute many threads concurrently. These threads are grouped physically into warps of 32 threads each. As stated GPUs contains many SMs, and these SMs can execute many threads concurrently. The threads in the SMs access data and instructions from the global memory (DRAM) at a given bandwidth. Bandwidth is best served by using as much fast memory and as little slow-access memory as possible. Thus, there exists on-chip memory such as registers which are allocated to individual threads, Read-only memory for specific tasks such as texture memory, and the L1 cache/shared memory, for fast data access within each SM.

The L1 cache/shared memory is on-chip memory that is shared within thread blocks or CUDA blocks. The shared memory usage is however controlled via software while L1 cache is controlled by hardware. Because L1 cache and shared memory exists on-chip, it is faster than both L2 cache and global memory. The L1 cache is however very small in size and not coherent. To ensure coherence in data accessed from the global memory by different SMs, the L2 cache is used.

L2 cache can be accessed by all threads in all CUDA blocks. Retrieving data from the L2 cache is faster than retrieving data directly from global memory (DRAM). In modern GPU architectures, access to global memory is cached in L1 and L2 by default.

3.1. L2 Caching Policy

The L2 cache potentially provides higher bandwidth and lower latency accesses to global memory. A typical cache line size is 128B in GPUs. In the Ampere architecture, the loads and stores can be serviced at 32 B granularity known as a sector[14].

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be persisting. On the other hand, if the data is only accessed once, such data accesses can be considered to be streaming.

A caching policy is used to determine which portion of data to cache. Due to the relatively small ratio of cache sizes to input data sizes of many HPC applications, there is the need for a policy to select the data to prioritize for caching. Consequently, the data access frequencies are considered in the eviction of the data cache lines. Streaming data for instance is first in the eviction priority order and will likely be evicted when cache eviction is required. This policy is known as the evict_first policy.

Data considered as persistent and hence stored in the set-aside region will be last in the eviction priority order and will likely be evicted only after other data with the evict_first and evict_normal policies are evicted.

The persistence offered by the evict_first 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.

3.2. L2 Cache Data Persistence Control

The A30 architecture offers a new feature that allows a portion of the L2 cache to perform persistent data accesses to device memory, which ultimately enables higher bandwidth and lower latency accesses to device memory.This is achieved through the use of APIs offered in the CUDA version 11 toolkit to set aside a portion of the 24-MB L2 cache to perform persistent data accesses to global memory. If this set-aside portion is not used by persistent accesses, normal accesses or streaming data can use it.

The L2 cache set-aside size for persisting accesses may be adjusted, within limits. For our experiments, we set aside a limit of 75% of the L2 cache memory for persisting accesses. Since the L2 cache size of the A30 GPU is 24MB, this translates to 18MB of L2 cache memory set aside for persistent accesses.

Ⅳ. Experimental Setup

4.1. Hardware and Software Description

Table [1] summarizes the experimental setup for our experiments. We execute our experiments on an Nvidia Ampere GPU device with 24GB of Device memory and 24MB of L2 cache. We use 1 of the 2 GPUs in our A30 environment. The compute capability is 8.0 and a cache line has a size of 128 Bytes. For profiling, we use Nvidia's Nsight Compute [17].

Table 1. Experimental Setup

GPU Device	NVIDIA A30
Compute capability	8.0
Device Memory	24GB
GPU memory bandwidth	933 GB/s
L2 cache size	24MB
L1 cache size	192KB
Profiler	Nsight Compute

4.2. Sliding Window Experiment

We implement a sample micro-benchmark [18] which uses a 1024 MB region in GPU global memory through the following kernel code Fig. 2.

```
__global__ void kernel(int *data_persistent, int *data_streaming, intdataSize, int freqSize)
{
int tid = blockIdx.x * blockDim.x + threadIdx.x;

data_persistent[tid % freqSize] = 2 * data_persistent[tid % freqSize];data_streaming[tid %
dataSize] = 2 * data_streaming[tid % dataSize];

}
```

Fig. 2. Kernel code for sliding window experiment [18]

An access policy window Fig. 3 specifies a contiguous region of global memory and a persistence property in the L2 cache for accesses within that region. As shown in Fig. 3, the stream level attribute data structure is used to set the region of the device memory which will persist in L2 cache when initially accessed.

```
cudaStreamAttrValue stream_attribute;                                    // Stream level attributes data structure
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr);  // Global Memory data pointer
stream_attribute.accessPolicyWindow.num_bytes = num_bytes;               // Number of bytes for persisting accesses.
                                                                         // (Must be less than Max Window Size)
stream_attribute.accessPolicyWindow.hitRatio  = 1.0;                     // Hint for persisting accesses in the num_bytes
region
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;  // Type of access property on cache hit
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;  // Type of access property on cache miss.

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

Fig. 3. Access Policy for sliding window experiment [18]

Similar to NVIDIA's sliding window experiment [18], we specified the access to the freqSize * sizeof(int) bytes of memory to be persistent and varied this persistent data region from 10MB to 40MB (Fig. 4) to model various scenarios where data fits in or exceeds the available set-aside portion of 18MB for our NVIDIA A30 GPU. Normal or streaming accesses can use the remaining 6MB of the non set-aside L2 portion. We used a fixed hit-ratio of 1.0 for our experiment.

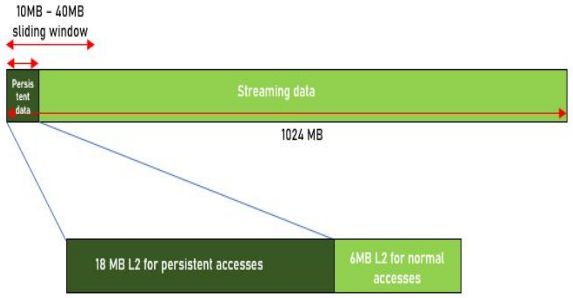

Fig. 4. Mapping the persistent accesses to L2 set-aside for our Nvidia A30 environment

The performance of the kernel in Fig. 2 is shown in Fig. 5 and Fig. 6. With a hit ratio of 1.0, the hardware attempted to cache the whole 40MB window in the set-aside L2 cache area. However, since the set-aside area (18MB) was smaller than the window, cache lines were evicted to make room for data required for the executions. The premature eviction of data before any significant reuse is known as cache thrashing.

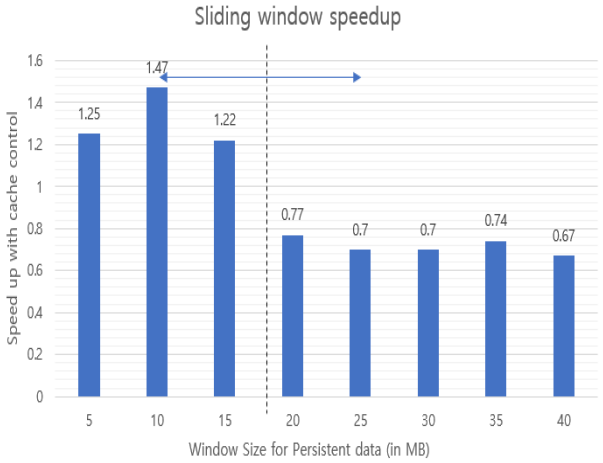

Fig. 5. Sliding window with hit-ratio

When the persistent data region fitted well into the 18MB set-aside portion of the L2 cache, a performance increase of as much as 47% is observed in Fig. 5. However, once the size of this persistent data region exceeded the size of the L2 set-aside cache portion, there was approximately 25% drop in performance. We attribute the fall in performance to thrashing of L2 cache lines.

Controlling the hit-ratio

The hitRatio value can be used to manually control the amount of data different accessPolicyWindows from concurrent CUDA streams can cache in L2. The hitRatio can therefore be used to reduce the amount of data moved into and out of the L2 cache.

In order to optimize the performance and reduce thrashing, we tuned the numbytes and hitratio parameters in the access window so that a random 10MB of the total persistent data (data_in_cache) was resident in the L2 set-aside cache portion and investigated the performance with an experiment. According to equation 1, this translated to a hit ratio of 0.556.

$$Hitratio = \frac{data\_in\_cache}{numbytes} \qquad (1)$$

From the results in Fig. 6, we observed an

overall improvement in performance for L2 set-aside cache portions which exceeded the size of the persistent data region.

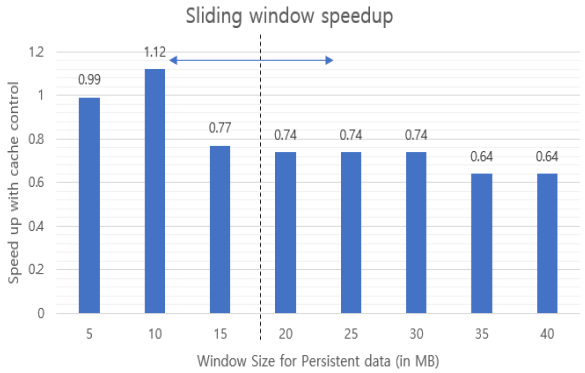

Fig. 6. Sliding window with adjusted hit-ratio

4.3. Workloads

We chose two workloads from the cudaSDKsamples[19] benchmark suites with data input sizes greater than our L2 cache size for our experiments. A short description of the workloads, the input data sizes and the kernels per workload is given below and summarized in Table 2.

Table 2. Description of workloads

Workload	Kernels and Functions	Input Data, MB
Histogram	Histogram64, mergedHistogram64	64
Conjugate Gradient	gpuConjugateGradient	108.8

Histogram: A histogram is a commonly used analysis tool in image processing and data mining applications. They show the frequency of occurrence of each data element [20]. The histogram used in our experiments has two (2) kernels: histogram64() and mergedhistogram64() used for analysis in our experiments.

Conjugate Gradient (CG) Solver: The conjugateGradientMultiBlockCG implements a conjugate gradient solver on a GPU using Multi Block Cooperative Groups. The sample used in our experiments has only one kernel [21].

4.4. Application Characterization for Frequent Accesses

We began by profiling each of the kernels for the workloads to determine the L2 cache access patterns. This was to inform the decision on the size of cache resources to be allocated to each application during scheduling. We considered the approach by Alsop et al. [22] and characterized the kernels according to the following:

Memory intensiveness: As a general rule, workloads with low compute bandwidth and high memory request bandwidth are more likely to be sensitive to caching policy than workloads with low memory request bandwidth and high compute bandwidth [22].

Frequent Data Accesses: Kernels with smaller data sizes generated from global memory into L2 cache but with relative large data sizes generated to L1 cache can be considered to be reusing data in the L2 cache compared to other kernels and are thus classified to have Persistent accesses. Kernels with similar or same data size generated from global memory into L2 cache and into L1 cache can be considered to have Streaming accesses. Finally, kernels considered to have Normal accesses are those with relatively smaller sized data generated into L1 cache compared to data generated from global memory into L2 cache.

V. Results

5.1. Application Characterization

Based on the values of the compute throughput and memory throughput for each of the kernels shown in Fig. 7, we ascertained that all the kernels were memory intensive with kernel histogram64 being the most memory intensive.

We also collected metrics on data request and access sizes to obtain insight into which

workloads have persistent accesses and which

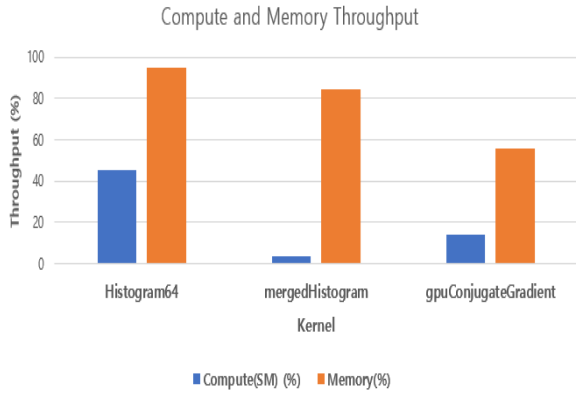

Fig. 7. The compute and memory throughput for the kernels

ones have streaming accesses. Based on the difference in the size of data generated in L1 relative to the data cached, we identified the access type for each kernels as either persistent, streaming or normal according to the caching policy used for Nvidia Ampere devices.

Table 3. Data transfer by the L2 cache

Kernel	Data Request MB	Data cache MB	Data to L1 MB	Access type
Histogram64	64	64	100	Streaming
mergedHistogram64	136.56	1.14	143.20	Persistent
gpuConjugateGradient	108.8	40.39	46.82	Normal

From Table 3, mergedHistogram64 was characterized to have persistent access type as it accessed the L2 cache more frequently. The data transferred through the L2 cache is also represented graphically in Fig. 8.

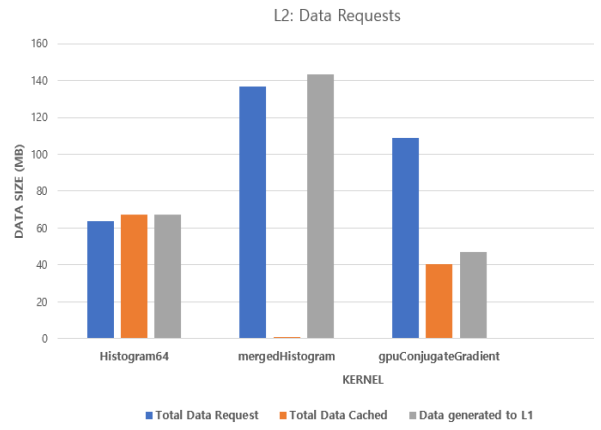

Fig. 8. Data transferred by the L2 cache

5.2. Warp Stalls

The warp scheduler can mask the delay of the warp by switching to a different warp when one warp is stopped owing to memory work or other reasons. With the Nsight Compute profiler, we collected warp stall sampling metrics for the first 100 address spaces during the execution of histogram and conjugated gradient kernels We compared the effect of allocating set-aside area to one of the kernels at a time during concurrent executions.

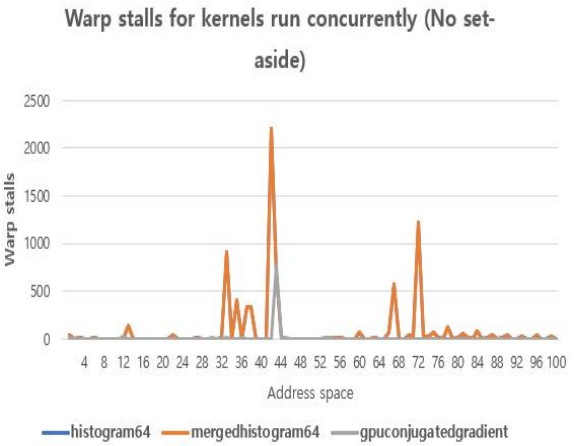

Fig. 9. Warp stalls for kernels run concurrently (no set-aside)

From the results in Fig. 9, we observed that, when there was no set-aside area in the L2 cache, histogram64 kernel did not have a warp stall. This confirmed the assertion that the data loaded into the L2 cache is hardly re-used.

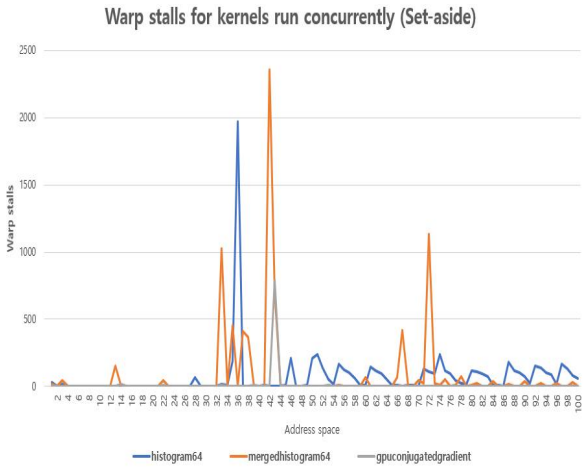

Fig. 10. Warp stalls for kernels run concurrently (set-aside)

On the other hand, when a set-aside area of 18MB is allocated to histogram64, the kernel had a bursty warp stall (Fig. 10).

NVIDIA maintains that normal or streaming accesses can use set-aside portions of the L2 when not in use however this may come at a cost of memory stalls.

5.3. Set-aside aware concurrent executions

The above observation for warp stalls revealed the need to identify the portion of L2 cache to allocate for persistence during concurrent executions in order to maximize the overall performance of the applications.

To investigate this, we considered three co-scheduling scenarios showing different allocations of L2 cache.

We set the hit ratio to 1.0 and the maximum persistence of the L2 cache to 0.75% of the total size of the L2 cache.

At the given hit ratio of 1.0, we tuned the window size of the applications for each concurrent run as follows: (H/mH=3, CG=15), (H/mH=9, CG=9) and (H/mH=15, CG=3). We observed the differentials in the number of elapsed cycles as shown in Table 4.

Table 4. Differentials in Elapsed Cycles for different set-aside areas

Kernel	H/mH=3 CG=15	H/mH=9 CG=9	H/mH=15 CG=3
Histogram64, H	−11312	16072	41104
mergedHistogram, mH	−12698	57302	−48538
gpuConjugateGradient, CG	51170	94080	115122

From the results, we observed that, because of the streaming nature of data accesses in histogram64, there was poor performance when 15MB of the L2 cache was reserved for persistent access by gpuConjugateGradient as the Histogram64 kernel had more elapsed cycles of (11312 cycles) relative to execution in single-mode.

On the other hand, there was a general increase in performance when the larger persistent data region (15MB) was allocated maximally to the histogram; the Histogram64 kernel had fewer elapsed cycles of about 41104 cycles relative to execution in single-mode.

The normalized speed-up values for the concurrent executions of kernels in both the histogram and conjugate gradient workloads according to the folllowing set-aside allocation (H/mH=3, CG=15), (H/mH=9, CG=9) and (H/mH=15, CG=3) is represented in Fig. 11.

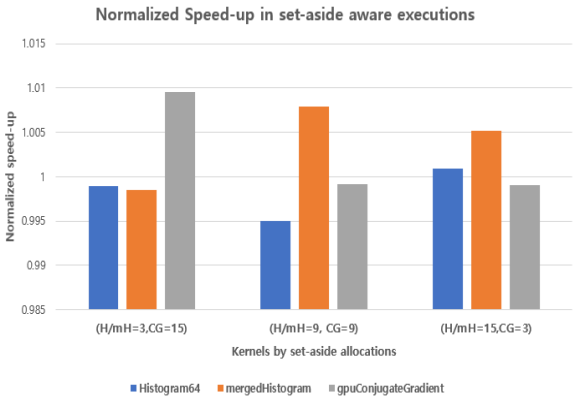

Fig. 11. Normalized speed-up in set-aside concurrent executions

From the results, we observed that, for optimal performance of all applications, larger persistent set-aside area must be allocated to kernels with streaming accesses such as histogram64, to enhance overall performance.

VI. Conclusions and Future Works

According to Alsop J. et al [22], although caching can significantly improve performance by enabling local data reuse, in some cases the best caching policy is not the one that enables the most caching.

With the new residency control feature introduced in Nvidia Ampere architectures, an end-user must decide on the best caching policy by identifying the access patterns of the workloads.

The kernel that generates the most data from the global memory may not necessarily be the kernel which requires persistence in the L2 cache. Contrary to the intuition to allocate less persistent data region to a kernel with streaming access whilst concurrently allocating more to the kernel with either normal or persistent data access, we observed that, allocating more persistent region to a kernel with streaming access when co-executed with that of normal access yielded optimal overall performance.

In future, we intend to expand the number of workloads profiled and observe their behavior for different co-scheduling scenarios.

References

[1] Nvidia Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programmingguide/index.html

[2] Tensorflow, https://www.tensorflow.org/tutorials/using_gpu

[3] LAMMPS, http://lammps.sandia.gov/doc/accelerate_gpu.html

[4] NCBI-BLAST, https://www.ncbi.nim.nih.gov/pmc/articles/PMC3018811

[5] GAUSSIAN, http://gaussian.com/relnotes/?tabid=2

[6] NAMD, http://www.ks.uiuc.edu/Research/namd/2.9/ug/node88.html

[7] VASP, http://www.vasp.at/index.php/news/44-administrative/115newreleasevasp541withgpusupport

[8] AMBER, http://ambermd.org/gpus/

[9] GAMESS, http://www.msg.ameslab.gov/gamess/versions.html

[10] Microsoft Azure, azure.microsoft.com/enus

[11] EC2 ELASTIC GPUS, https://aws.amazon.com/ec2/ElasticGPUs/,2017.

[12] 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)

[13] Rogers, T.G, O'Connor, M.and Aamodt, T.M., "Cache-conscious wavefront scheduling", Proceedings of the 45th annual IEEE/ACM International symposium on microarchitecture, MICRO (2012)

[14] Lal, S., Sharat Chandra Varma, B., and Juurlink, B. (2022), "A Quantitative Study of Locality in GPU Caches for Memory Divergent Workloads" International Journal of Parallel Programming, 50, 189216. https://doi.org/10.1007/s10766022007292

[15] 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

[16] 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.

[17] NSIGHT COMPUTE, https://developer.nvidia.com/nsight-compute

[18] Sliding Window Experiment, https://docs.nvidia.com/cuda/cuda-c-best-practicesguide/index.htmlmemory-optimizations

[19] CUDA SAMPLES https://github.com/NVIDIA/cudasamples/

[20] Nvidia developer downloads, https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf

[21] CUDA Samples Documentation, https://docs.nvidia.com/pdf/CUDA_Samples.pdf
[22] J. Alsop et al., “Optimizing GPU Cache Policies for MI Workloads,” 2019 IEEE International Symposium on Workload Characterization (IISWC), 2019, pp. 243-248, doi:10.1109/IISWC47752.2019.9041977.
[23] N. Duong et al., “Improving cache management policies using dynamic reuse distances”, in Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012

테오도라 아두푸 (Theodora Adufu)

2013년 : 가나 대학교 컴퓨터과학 및 경제학과 졸업(학사)
2016년: 숙명여자대학교 컴퓨터과학과 졸업(석사)
2022년～현재: 숙명여자대학교 컴퓨터과학과 (박사)

<관심분야> 컨테이너, 클라우드 컴퓨팅, HPC 클라우드, Accelerated Computing(GPUs)

김 윤 희 (Yoonhee Kim)

1991년 : 숙명여자대학교 전산학과 졸업(학사)
1996년 : Syracuse University 전산학과 졸업(석사)
2003년: Syracuse University 전산학과 졸업(박사)
1991년～ 1994년 한국전자통신연구원 연구원.
2000년~2001년 Rochester Institute of Technology 컴퓨터공학과 조교수.
2001년 ～ 2016년 숙명여자대학교 컴퓨터과학부 교수.
2017년 ～ 현재 숙명여자대학교 소프트웨어학부 교수.

<관심분야> 클라우드 컴퓨팅, 워크플로우 제어, HPC 클라우드, Accelerated Computing(GPUs)