IEIE Transactions on Smart Processing and Computing, vol. 4, no. 2, April 2015
http://dx.doi.org/10.5573/IEIESPC.2015.4.2.078

Study of Cache Performance on GPGPU

Kyu Hyun Choi and Seon Wook Kim

Department of Electrical and Computer Engineering, Korea University / Seoul, South Korea {nurlonn, seon}@korea.ac.kr

***** Corresponding Author: Seon Wook Kim

Received November 20, 2014; Revised December 27, 2014; Accepted February 12, 2015; Published April 30, 2015

** Short Paper*

***Abstract*:** General-purpose graphics processing units (GPGPUs) provide tremendous computational and processing power. Despite the latency hiding mechanism, a GPU architecture requires high memory bandwidth and lower latency between computational units and the memory system. For this reason, the current GPU architecture has private L1 caches in each core and a shared L2 cache to increase performance by reducing memory latency. But in some cases, this CPU-like cache design is not suitable for GPGPUs. In this paper, we analyze detailed cache performance related to GPGPU application characteristics, and suggest technical alternatives for the GPGPU architecture as future work.

***Keywords*:** GPGPU, Cache, Performance analysis, Application characteristics

1. Introduction

General-purpose graphics processing unit (GPGPU) computing has been widely used to accelerate heavy applications. The Compute Unified Device Architecture (CUDA) [1] and Open Computing Language [2] frameworks provide convenient programming environments for GPUs to solve many complex computational problems.

Prior research work suggests that memory bandwidth is the bottleneck for GPU performance, even if GPU memory has a much higher memory bandwidth than CPU memory. So GPUs use address coalescing to reduce memory requests, and have memory latency hiding mechanisms, like the context switch. But memory bandwidth is still limited. To resolve the bandwidth problem, many commercial GPUs have small L1 and L2 static random access memory (SRAM) caches to leverage on-chip data reuse. As a result, the performance of GPGPU programs depends on cache efficiency. However, in our observations, current cache design is inefficient when running memory-intensive applications..

A number of studies have attempted to optimize application algorithms to GPU memory systems [3]. Most of them fit their algorithms to the GPU cache to maximize cache efficiency, and could attain performance improvement. But most of them are compute-intensive applications. Furthermore, the working set of GPGPU applications has become larger and larger to exploit maximum parallelism. In this case, cache performance is hampered by thrashing due to massive multithreading.

In this paper, we present details on GPU cache performance depending on application characteristics. The observations made in this paper will provide useful guidance for directing future architecture and software research. The rest of this paper is organized as follows. Section 2 gives our experimental setup. Section 3 shows the detailed analysis results. Finally, Section 4 draws the conclusion.

2. Backgrounds

A GPU consists of a number of simple in-order and data-parallel cores called shader cores. Shader cores have single-instruction, multiple thread (SIMT) lanes, and execute a kernel code in lock-step. In the CUDA programming model, the kernel comprises thousands of scalar threads. The group of threads is called a cooperative thread array (CTA). The GPU assigns CTAs to shader cores. After the CTA assignment, the GPU core divides a CTA into warps. A warp is a basic unit of execution that contains 32 scalar threads. A warp scheduler inside the shader core selects ready warps every four cycles and issues them to the execution pipelines in a loose, round robin fashion. The scheduler skips non-ready warps, such as those waiting on global memory access.

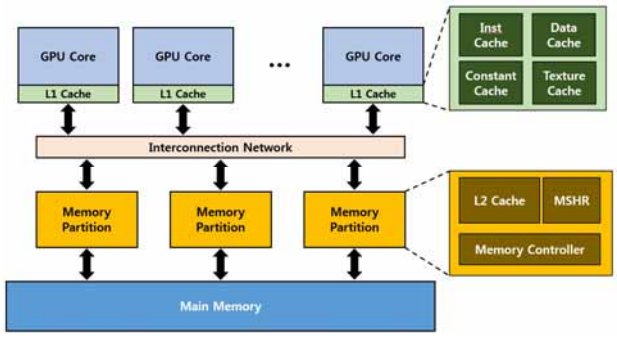

Fig. 1. GPU memory system.

In other words, whenever any thread inside a warp faces a long latency operation, the whole warp is taken out of the scheduling pool until the long latency operation is over. Meanwhile, other warps that are not waiting are sent to the execution pipeline in round robin order. There are many warps running on each shader core; thus, a shader core can maintain throughput and hide high memory access latency. Furthermore, threads in the warp try to access contiguous memory regions, so the GPU combines memory requests to make a new request. This is the memory coalescing.

Fig. 1 shows a GPU memory system. Each GPU core has a private L1 I/D cache, a read-only constant cache and a texture cache, and low-latency scratchpad memory. Entire cores share L2 cache and dynamic random access memory (DRAM). GPU cores and L2 caches are connected via an interconnection network. In our system, the cache is non-inclusive non-exclusive. And there is no cache coherence protocol, so the L1 data cache has a two-write policy: write-through for global memory, and write-back for local memory.

3. The Proposed Scheme

We used GPGPU-SIM v.3.2.2 [4] for our performance analysis. The simulator mimics a generic NVIDIA Fermi GPU and provides a cycle-accurate simulation environment. The detailed configuration is shown in Table 1, and we chose our test benchmark from Rodinia [5], NVIDIA SDK [6], and some other benchmarks [7, 8].

4. Detailed Cache Performance Analysis

Fig. 2 shows the L1 and L2 cache miss ratio. On average, the L1 data cache miss ratio of a GPU is 82.3%, whereas that of a CPU is 6.3% to 33% [9]. Fig. 3 is the L1 cache hit ratio depending on LRU position. The hit ratio is the ratio of L1 cache hit counts at each position over total L1 access counts. Fig. 4 is the L2 cache hit ratios depending on LRU position. They are the ratios of L2 cache hit counts at each position over the total L2 access counts. Figs. 5 and 6 are L1 and L2 cache access classifications by access characteristics, global memory read/write, local memory read/write, and accesses from

Table 1. GPU configuration for our work.

Parameters	Configurations
Core configuration	**15 shader cores, 1,400MHz, SIMT width = 32**
Resources / Core	Max. 1536 threads, 48KB shared memory, 32K registers
L1 caches	16KB 4-way L1 Data cache 2KB 4-way L1 Inst cache 12KB 24-way L1 Tex cache 8KB 2-way Const cache 128B Block size
L2 caches	1,536KB 16-way L2 shared cache 128B Block size

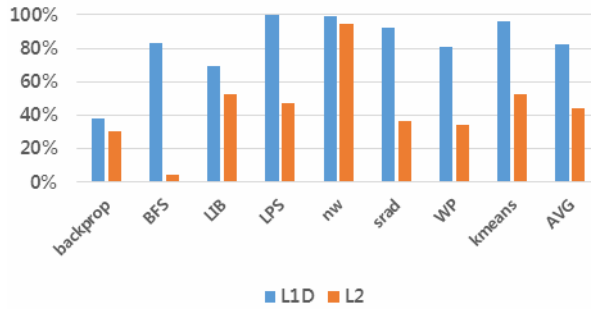

Fig. 2. GPU cache miss ratio.

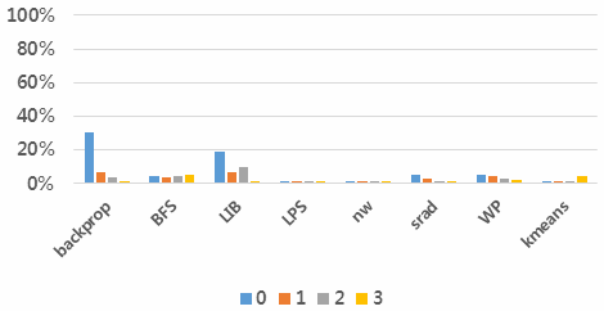

Fig. 3. L1 cache hit ratio depending on LRU position.

other L1 caches, such as the instruction cache and texture cache. Global memory access means that GPU threads access the global memory space shared by all other GPU cores. Local access is where the GPU thread accesses threads in private local memory space. Hit means L1 data cache hit counts, and miss is L1 data cache miss counts. Hit reserved is the total number of pending hits in the cache. A pending hit access has hit a cache line in a reserved state, which means an inflight memory request was already sent by a previous cache miss on the same line. This access can be merged with that previous memory access so it does not increase memory traffic. Note that a hit reserved is treated as a cache miss in Figs. 2-4.

4.1 Backprop

Backprop is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. It has a relatively low miss ratio on both L1 (0.38) and L2 (0.30). It reads the value twice from same

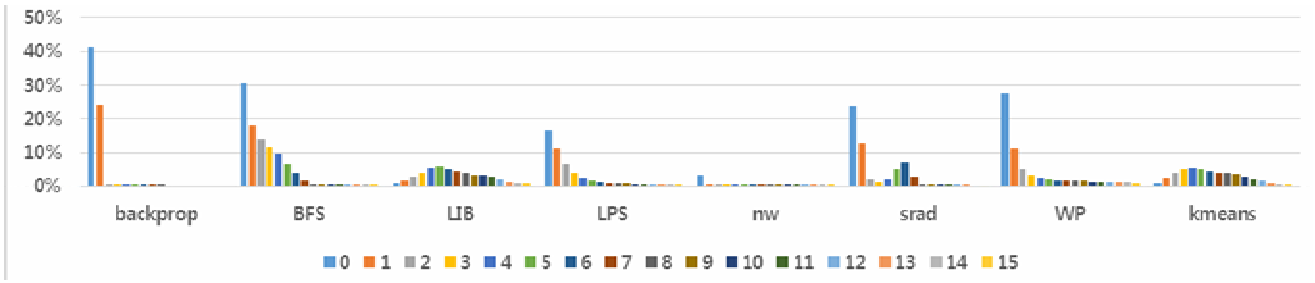

Fig. 4. L2 cache hit ratio depending on LRU position.

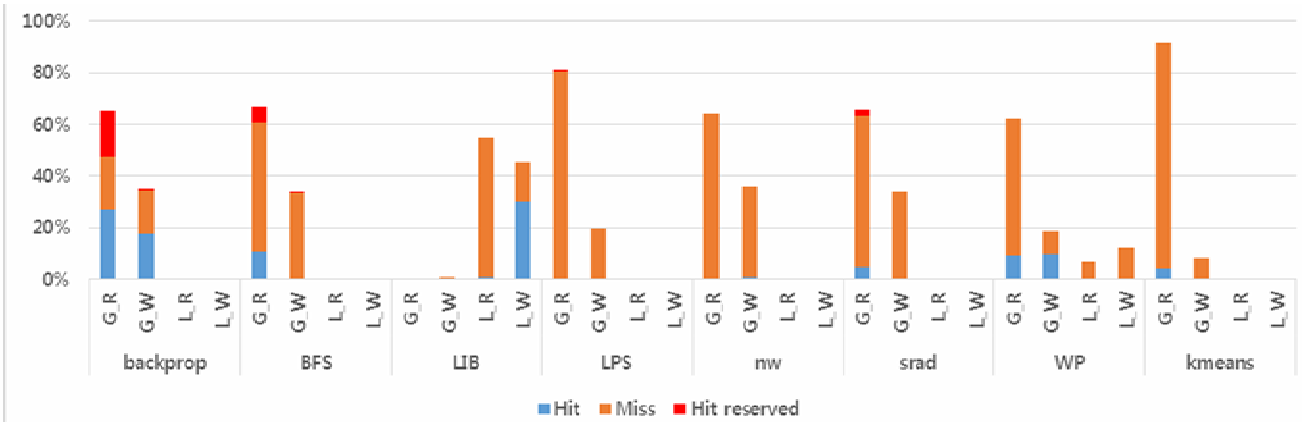

Fig. 5. L1 cache access classifications (G = Global, L = Local, R = Read, W = Write).

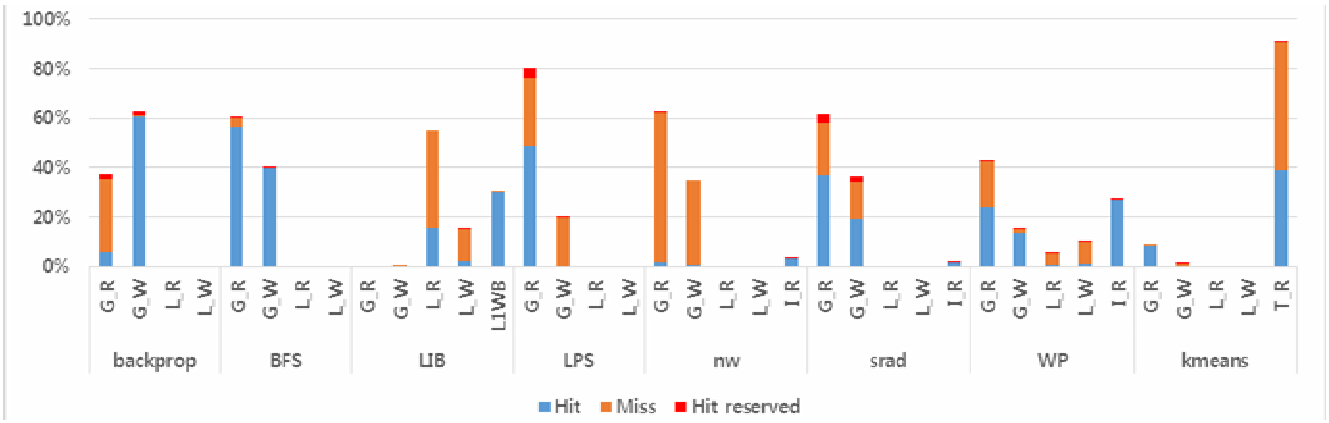

Fig. 6. L2 cache access classifications (G = Global, L = Local, I = Instruction, T = Texture, R = Read, W = Write, L1WB = Write Back request from L1 cache).

global memory address repeatedly. So the second load could get the value from the MRU position of the L1 cache. The kernel compiler does not optimize this redundant load instruction because of memory consistency. Global memory is shared for all other GPU cores, and other GPU cores could change the value. In this application, other cores do not modify the value. So, we can remove second load request and get the value from the register. This optimization could lead to slight performance improvement for overall GPU cores, but does not affect cache efficiency.

4.2 BFS

The breadth first search (BFS) algorithm on a graph has a high L1 cache miss ratio (0.83) and a low L2 miss ratio (0.03). Many GPU applications have a regular memory access pattern because they access memory by thread ID and block ID. But in this application, each GPU thread travels a data node separately, so the loading address is determined by the value of previous nodes. The irregular memory access pattern spoils L1 cache, and causes a high L1 cache miss ratio. On the other hand, the L2 miss ratio is quite small because the data size is much

smaller than the L2 cache size. We use an input file that has 65,536 nodes, and each node takes 8 bytes. So only one-third of the L2 cache space is used. When we use larger data sizes that do not fit in the L2 cache, the L2 miss ratio will drastically increase.

4.3 LIB

LIB carries out a Monte Carlo simulation. LIB has a moderate L1 (0.69) and L2 miss ratio (0.52). This application does not use global memory. Each thread loads and stores values in its private local array. The application uses constant cache to reuse some scalar variables. But local memory accesses are much more than constant cache accesses. The access patterns of each thread are regular, but hundreds of threads emulatively read their own local array. So both hit position graphs are uneven. The L1 hit ratio is moderate, but most of them are L1 write hits. Even if write hits reduced additional traffic, the performance impact is quite a bit lower than read hits. Data reuse is partially achieved in the L2 cache, but every thread has its own private local array and competitively uses it. So cache pollution causes L1 read misses and a lower L2 cache hit ratio.

4.4 LPS

The 3D Laplace solver (LPS) application has an extremely high L1 miss ratio (0.99) and a moderate L2 miss ratio (0.47). This application makes a huge number of shared memory accesses. L1 data cache access is about 8% of shared memory access. Shared memory is scratchpad memory that is shared all over the CTA and does not affect the cache hierarchy. The global memory access pattern is slightly irregular, because the addresses of global access diverge depending on the position of the threads. Furthermore, the working set size is much bigger than the L1 cache size. For this reason, LPS suffers an extremely high L1 cache hit ratio. In the L2 cache, the slight irregular memory access pattern increases the L2 cache hit ratio due to the spatial locality. So a larger cache is needed for this application.

4.5 NW

Nw is a global optimization algorithm for DNA sequence alignments. It has an extremely high miss ratio in both L1 (0.99) and L2 caches (0.94). This application does not exploit thread level parallelism in the GPU. A limited amount of CTA is executed at the same time. And data reuse does not occur in both caches. All global data are used once and thrown away. Most L2 cache hits come from the instruction fetch. In this case, any kind of cache optimization technique is completely pointless.

4.6 SRAD

Srad is a diffusion algorithm to remove noise in ultrasonic and radar imaging applications based on partial differential equations. It has a high L1 miss ratio (0.92) and a moderate L2 miss ratio (0.36). Due to the programmer's optimization, the working set size per CTA is smaller than the L1 cache size. The working set size of a CTA is a 16-by-16 two-dimensional floating point matrix. A cache line can contain two CTA matrix elements of the x-axis. But multiple CTAs are executed at the same time in a GPU core, so data thrashing due to conflict misses will reduce cache efficiency. When we use a 16-way 16KB L1 cache, the L1 miss ratio is reduced to 0.78. In this case, even though most data are evicted by thrashing, the first element remains in the LRU position and can be reused. On the other hand, the L2 cache could contain evicted data from the L1 cache due to the L1 conflict miss. For this reason, the L2 cache has a relatively low miss ratio.

4.7 WP

WP is a numerical weather prediction application. It has a high L1 cache miss ratio (0.80) and a low L2 cache miss ratio (0.34). Like the LIB application, each thread individually works with its iteration number and does not use a thread ID. The application stores values in registers to reduce redundant memory accesses. But this requires a huge number of registers and limits thread level parallelism. Furthermore, based on the application's purpose, the WP kernel uses various kinds of input data. Registers could not cover all global and local memory accesses. So WP has a high L1 miss ratio. Some L1 hits and L2 hits are caused by spatial locality, but generally, the access pattern is not suitable for cache memory. And about 40% of the L2 cache hits come from an instruction fetch, because the kernel code size is bigger than the L1 instruction cache.

4.8 Kmeans

Kmeans is a clustering algorithm used widely in data mining. It divides data objects into sub-clusters, and finds the mean values in the sub-clusters. Kmeans uses both constants and texture caches. It loads some scalar values from constant cache, and loads most of the data from texture cache. In this application, the access count of the data cache is only 6% of total cache access counts. Constant cache has a very low miss ratio (< 0.001), but texture cache and data cache have very high miss ratios (0.95, 0.96). The working set is much bigger than the texture cache size, so data thrashing occurs in texture cache. The data from the data cache is rarely used only once, so the data cache has a high miss ratio. In L2 cache, the data size is still much bigger than the L2 cache size, but the working set fits into L2 cache. So the L2 cache miss ratio is relatively low (0.52) compared to the L1 cache miss ratio. Most global reads are hits in L2 cache because of spatial locality, but a huge number of texture memory accesses from thousands of threads makes conflicts miss each other.

5. Conclusion

In this paper, we analyze each GPU application one by one and discuss GPU cache efficiency. According to our analysis, a small L1 data cache is not efficient for GPU

applications. To exploit maximum performance, a GPU cache needs much more capacity. But SRAM cache is energy hungry and has limits on capacity. Our conclusion is that SRAM GPU cache should only be used for small on-chip scratchpad memory, such as shared memory and constant cache, and the future of the GPU cache will be die-stacked DRAM cache. Research into die-stacked DRAM cache is our future work..

Acknowledgement

This work was supported by the Industrial Strategic Technology Development Program (10041664, The Development of Fusion Processor based on Multi-Shader GPU) funded by the Ministry of Trade, industry & Energy (MI, Korea)

References

[1] NVIDIA, “CUDA C Programming Guide,” Oct. 2010. Article (CrossRef Link)
[2] KHRONOS Group, “The OpenCL Specification,” Aug 2012. Article (CrossRef Link)
[3] Govindaraju, Naga K., et al. "A memory model for scientific algorithms on graphics processors," In Proceeding of the 2006 ACM/IEEE Conference on Supercomputing, Nov 2006. Article (CrossRef Link)
[4] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator," IEEE International symposium on Performance Analysis of Systems and Software (ISPASS), pp. 163-174, April 2009. Article (CrossRef Link)
[5] Che, Shuai, et al. "Rodinia: A benchmark suite for heterogeneous computing," IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct 2009. Article (CrossRef Link)
[6] NVDIA, "CUDA C/C++ SDK Code Samples," 2011. Article (CrossRef Link)
[7] Harish, Pawan, and P. J. Narayanan. "Accelerating large graph algorithms on the GPU using CUDA," International Conference on High Performance computing, Springer Berlin Heidelberg, pp. 197-208, 2007. Article (CrossRef Link)
[8] Michalakes, John, and Manish Vachharajani. "GPU acceleration of numerical weather prediction," Parallel Processing Letters, vol. 18, no. 5, pp.531-548, 2008. Article (CrossRef Link)
[9] Phansalkar, Aashish, Ajay Joshi, and Lizy K. John. "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," ACM SIGARCH Computer Architecture News, vol. 35, no. 2, pp. 412-423, 2007. Article (CrossRef Link)

Kyu Hyun Choi received his BS in Electrical Engineering at Korea University, Seoul, Korea, in 2012. Currently, he is a PhD student at Korea University. His research interests include microarchitectures, GPUs, parallel computing and memory systems design.

Seon Wook Kim received a BS in Electronics and Computer Engineering from Korea University, Seoul, Rep. of Korea, in 1988. He received an MS in Electrical Engineering from Ohio State University, Columbus, Ohio, USA, in 1990, and a PhD in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana, USA, in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Inter/KSL from 2001 to 2002. Currently, he is a professor with the School of Electrical and Computer Engineering at Korea University and is Associate Dean for Research at the College of Engineering. His research interests include compiler construction, microarchitectures, and SoC design. He is a senior member of ACM and IEEE.

Copyrights © 2015 The Institute of Electronics and Information Engineers