JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 11, No. 2, June 2017, pp. 69-77

Enhancing GPU Performance by Efficient Hardware-Based and Hybrid L1 Data Cache Bypassing

Yijie Huangfu and Wei Zhang*

Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
huangfuy2@vcu.edu, wzhang4@vcu.edu

Abstract

Recent GPUs have adopted cache memory to benefit general-purpose GPU (GPGPU) programs. However, unlike CPU programs, GPGPU programs typically have considerably less temporal/spatial locality. Moreover, the L1 data cache is used by many threads that access a data size typically considerably larger than the L1 cache, making it critical to bypass L1 data cache intelligently to enhance GPU cache performance. In this paper, we examine GPU cache access behavior and propose a simple hardware-based GPU cache bypassing method that can be applied to GPU applications without recompiling programs. Moreover, we introduce a hybrid method that integrates static profiling information and hardware-based bypassing to further enhance performance. Our experimental results reveal that hardware-based cache bypassing can boost performance for most benchmarks, and the hybrid method can achieve performance comparable to state-of-the-art compiler-based bypassing with considerably less profiling cost.

Category: Embedded computing

Keywords: GPU; CUDA; Cache bypassing; Memory traffic; Profiling

I. INTRODUCTION

Graphics processing units (GPUs) have become increasingly popular for general-purpose computing on GPUs (GPGPUs). Modern GPUs can support massive parallel computing with thousands of cores and extremely high-bandwidth external memory systems. Leading GPU vendors like NVIDIA and AMD have released software development kits such as Compute Unified Device Architecture (CUDA) and OpenCL, allowing programmers to use C-like programming language to write general-purpose code for execution on GPUs.

To accommodate the data locality in general-purpose high-performance applications, major GPU vendors have introduced cache memory in conjunction with the shared memory to benefit a wide variety of GPGPU applications. For example, L1 data cache and the unified L2 cache are included in NVIDIA Fermi and Kepler architectures, in which the sizes of the L1 data cache and shared memory are configurable while the aggregate on-chip memory size is fixed. Although cache memory can effectively hide access latency for data with good temporal and/or spatial locality for CPUs and GPUs, GPGPU applications may exhibit divergent memory access patterns from traditional CPU applications. For example, recent study reveals that GPU caches have counter-intuitive performance tradeoffs [1]. Moreover, due to the large number of threads available, the GPU L1 data cache is shared by all threads running on the same SM, enabling data with high reuse potential to be evicted by other data with no or low reuse potential. Therefore, it is critical to explore techniques to exploit on-chip cache memory effectively to

Open Access http://dx.doi.org/10.5626/JCSE.2017.11.2.69 http://jcse.kiise.org

Received 21 April 2017; **Accepted** 02 June 2017

*Corresponding Author

pISSN: 1976-4677 eISSN: 2093-8020

boost GPU performance and/or energy efficiency. Specifically, for embedded and mobile GPU applications, it is also crucial to develop cost-effective optimization methods for boosting performance and/or energy efficiency.

Cache bypassing is a technique to reduce cache pollution by not storing data with no or low locality into the cache. Due to the relative small size of the GPU L1 data cache as compared to the large amount of data size of a typical GPGPU program, cache bypassing is especially attractive to GPUs. Recently, Xie et al. [2] proposed a compiler-based GPU cache bypassing framework, in which each load instruction is set to be with the *ca* or *cg* cache operator, based on light-weight profiling. By default, on the CUDA platform, global memory access is cached in L1 and L2 caches (with the compilation flag of *-Xptxas -dlcm=ca).* The data can also be configured to be cached only in the L2 cache *(-Xptxas -dlmc=cg)* [3]. This approach, however, incurs significant cost to profile the GPU cache and memory access patterns, and to develop the optimization plan. In addition, the program must be recompiled to enable the bypassing, which can impede its use on real applications.

To address this problem, this paper proposes to develop a pure hardware-based cache bypassing method for GPU applications, which can be applied to any GPU code at run time. In addition, we also study a hybrid method to combine the profiling knowledge and the hardware-based bypassing to enable more intelligent cache bypassing applications without incurring significant costs as the compiler-based approach [2].

II. BASELINE GPU ARCHITECTURE

A GPU typically consists of an array of highly threaded streaming multiprocessors (SMs), and each SM has many streaming processors (SPs) that can execute threads in parallel. When a GPU kernel is launched, the runtime creates massive concurrent GPU threads organized hierarchically. Many threads (32 in NVIDIA GPU) with consecutive IDs compose a warp (or wavefront), multiple warps form a thread block, and all thread blocks compose a grid. A warp is the unit in GPU scheduling; in which all threads proceed in a lockstep fashion.

The L1 data cache and the shared L2 cache are included in the CUDA devices with compute capability of version 2.0 and higher. Each CUDA SM has its own L1 data cache, as shown in Fig. 1. The size of the L1 data cache can be configured to be either 16 kB or 48 kB, and the L2 unified cache has the size of 768 kB in the Fermi architecture [4]. In Kepler, the L1 data cache can also be configured to 32 kB as well. The L1 data cache can be enabled or disabled at compile time.

When the L1 data cache is enabled, the load instructions access data through the L1 data cache, while the store instructions write through the data to the L2 cache and the corresponding copy in the L1 data cache is invalidated [1]. Since the stored instructions bypass the L1 data cache by default, we focus on studying the load instructions in this work.

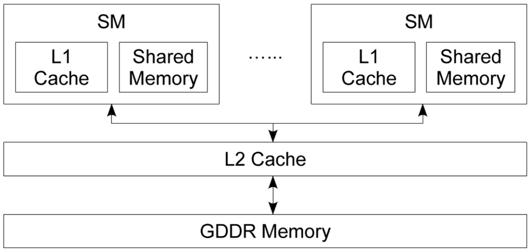


Fig. 1. GPU memory system with L1 and L2 caches.

III. MOTIVATION

A. Global Load Utilization Rate

The 32 threads in a warp access the global memory in a coalesced pattern. Assuming that each thread needs to fetch 4 bytes, if the data needed by each thread are well coalesced, this load operation can be serviced by one 128-byte transaction, as shown in Fig. 2(a). In this case, all the data in the memory transaction are useful, thus the utilization rate (or efficiency) of this load, which represents the percentage of bytes transferred from global memory that are used by the GPU, is 100% (128/128). However, when the memory access pattern changes slightly, as shown in Fig. 2(b) and 2(c), the address range becomes 96 to 223, which spans across the boundary of 128 bytes. In this case, two 128-byte transactions are needed to transfer data needed by threads. Therefore, the

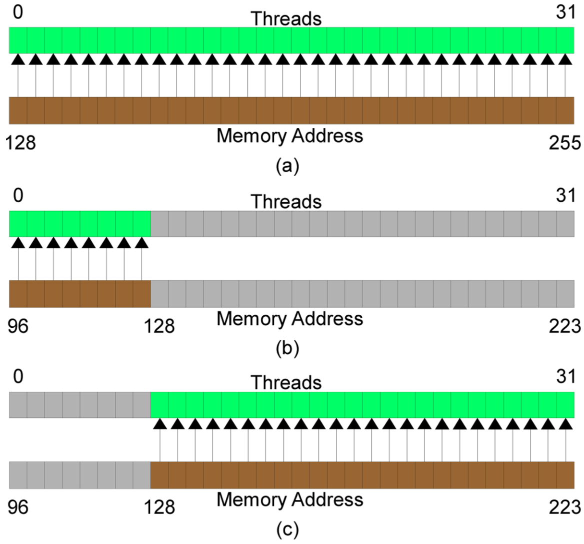


Fig. 2. Examples of different memory access patterns that lead to various global load utilization rates.

utilization rates of these two transactions are 25% and 75%, respectively, resulting in a 50% (128/256) overall utilization rate. This indicates half of the memory traffic, generated by these two load operations, are useless and unnecessary if they are not reused, which may decrease performance and energy efficiency for GPGPU computing.

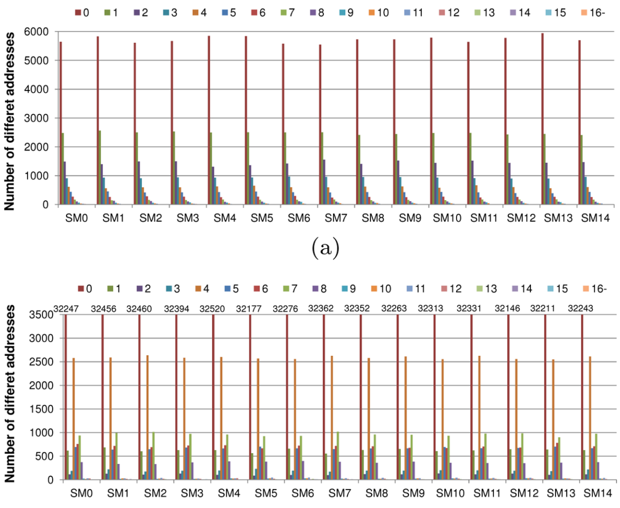
The example of low load utilization rates in Fig. 2 may be caused by improper mapping between threads and memory addresses, which, sometimes but not always, can be avoided by programmers. However, divergences in the CUDA kernel, which are caused by algorithms and are generally difficult to eliminate, can also lead to such load operations with low utilization rates.

Moreover, in the CUDA programming model, if the required data are cached in L1 and L2 data caches, memory access is done by 128-byte transactions. However, if the data are only stored in L2 cache (i.e., bypassing the L1 data cache), 32-byte transactions are used instead [3]. Therefore, for the load operations depicted in Fig. 2(b) and 2(c), assuming the data are not reused from the cache, using only 32-byte transactions can reduce over-fetching of useless data and therefore reduce memory traffic. Bypassing the L1 data cache for such kinds of load operations may be a better choice to reduce memory bandwidth pressure and to attain better performance. At the same time, since the L1 data cache is small and is shared by all the SPs in an SM, bypassing the L1 data cache may leave more space to store data with higher utilization rates or locality, which can reduce cache pollution and therefore benefit performance.

B. Data Reuse Times in GPU L1 Data Cache

The GPGPU applications usually operate on a massive volume of data. However, cache line usage among data with different addresses may differ significantly. This is not only because GPGPU applications can exhibit irregular data access patterns, but also because effective L1 data cache space per SP is too small. Therefore, even if some data are reused within a warp, such data may have been replaced from the cache by other data from the same warp or from other warps from the same thread block before they can be reused, resulting in cache misses and hence increasing global memory access.

Fig. 3 shows the data reuse distribution in the L1 data cache across different SMs for the benchmarks *gaussian* and *srad,* both of which are selected from *Rodinia* benchmark suite [5]. The experimental configuration and the evaluation methodology are detailed in Section V. In this figure, each bar indicates the number of different data addresses that are reused in the L1 data cache by a certain number of times, which varies from 0, 1, up to 15, or more. As we can observe, the number of different addresses reused in the L1 data cache varies slightly across different SMs because of the GPU's SIMD execution model. We also find for both benchmarks a considerable volume of data addresses are never reused or are only reused for a limited number of times. For example, in *gaussian*, nearly half of the addresses are used for just once, while in the *srad* most of the addresses are not reused at all. The considerably low temporal locality from GPGPU applications is quite different from typical CPU applications that tend to have good temporal locality; therefore, we must explore novel cache management techniques for GPUs.



(a)

(b)

Fig. 3. The data reuse distribution in the L1 data cache. (a) Data reuse distribution of *gaussian* benchmark. (b) Data reuse distribution of *srad* benchmark.

For data that are never reused, loading the data into the cache is helpful to reduce neither latency nor memory bandwidth. To the contrary, bypassing them may reduce cache pollution. Even if the data are reused a few times, loading them into the L1 data cache may increase global memory traffic if load utilization rate is low. This may negate the benefit of a small number of cache hits. Therefore, it becomes viable to bypass those data that are never reused or only reused a few times to reduce memory bandwidth pressure and cache pollution for GPUs.

IV. GPU L1 DATA CACHE BYPASSING

A. Bypassing by Addresses vs. by Instructions

By default, on the CUDA platform, global memory accesses are cached in L1 and L2 caches (with the compilation flag of *-Xptxas -dlcm=ca).* The data can also be configured to be cached only in the L2 cache *(-Xptxas -dlmc=cg)* [3]. The *ca* and *cg* cache operators for memory load instructions are used to support such configurations [6]. Such a mechanism can be used to implement cache bypassing by compilers.

While Xie et al.'s approach [2] can automatically ana-

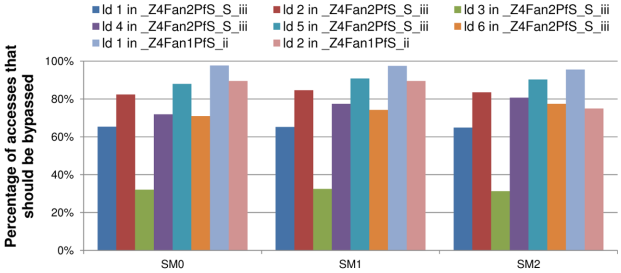


Fig. 4. Percentage of accesses that should be bypassed of each load instruction in the kernels of *gaussian* benchmark.

lyze the GPU code and select global load instructions for cache access or bypassing intelligently, enabling cache bypassing decisions based on each global load instruction may lead to suboptimal results. This is because each global load instruction can access a wide range of data addresses that can have distinct reuse behavior. Bypassing at the load instruction level is only effective if the data accessed by load instruction have uniform locality. For instructions that access data with diverse reuse characteristics, enabling bypassing decisions at the instruction level is too coarse-grained, i.e., it can only choose between bypassing or caching all data accessed by this instruction, not according to the subset of data access with different temporal and spatial locality. To address this deficiency, in this paper, we study an address based cache bypassing method that enables the GPU to bypass data at finer granularity.

Fig. 4 shows, according to each load instruction of the benchmark *gaussian*, the percentage of the memory access to different addresses that have low L1 cache line reuse time and low load utilization rate (see Eq. (1) in Section IV-B for details).

We focus on analyzing 8 load instructions in 2 kernels (Id 1-6 of kernel *_Z4Fan2PfS_S_iii* and Id 1-2 of kernel *_Z4Fan1PfS_ii)* from *gaussian.* We find that each global load instruction has a varying fraction of data accesses that should or should not be bypassed. For example, 65.3% of data accessed by ldl from kernel *_Z4Fan2PfS_S_iii* should be bypassed, indicating 34.7% of data accessed by this load should not be bypassed. Therefore, simply bypassing all the access from one load instruction will cause performance overhead for those data access that should not be bypassed. Similarly, simply caching all access from one load instruction will lose the performance improvement opportunity for those data access that should be bypassed. To facilitate finer-grained control of cache bypassing, in this study, we chose to implement the GPU cache bypassing based on individual data addresses instead of the load instructions.

B. Heuristic for GPU Cache Bypassing

We propose to use profiling to identify LI data cache access that should be bypassed. We focus on bypassing the data accesses that have low load utilization rates and low reuse times in the LI data cache, with the objective of minimizing global memory traffic. More specifically, for each data address A that is accessed by a global load, we use profiling to collect its load utilization rate U and the reuse time R. We use Eq. (1) to determine which data access should be bypassed.

$$U \times (1 + R) < 1 \qquad (1)$$

In the above equation, $(1 + R)$ represents the number of times A is accessed from the LI data cache, including the first time when it is loaded into the cache, i.e., 128 bytes are transferred from global memory. If U is 1, then this product is at least 1, even if A is not reused at all, indicating A should not be bypassed. On the other hand, if U is less than 1, and if R is 0 or a small integer (e.g., 1, 2, 3) such that the condition in Eq. (1) holds, then storing A into the LI data cache will increase global memory traffic as compared to bypassing this access from the LI data cache. Therefore, in this case, bypassing A can reduce global memory traffic, potentially leading to better performance or energy efficiency. The reduction of cache pollution will also be a positive side effect of bypassing this data from the LI data cache.

Our cache bypassing method considers spatial locality (i.e., U) and temporal locality (i.e., R). For example, for the memory access pattern with low load utilization rate as depicted in Fig. 2(b), i.e., U = 25%, this address must be reused at least three times in the LI data cache (i.e., $R \geq 3$) to not be bypassed. In contrast, for the memory access pattern with high load utilization rate that is shown in Fig. 2(c), i.e., U = 75%, if this address is reused at least once from the LI data cache (i.e., $R \geq 1$), then it should not be bypassed.

To support the profiling-based method, we modified the *GPGPU-Sim* [7] by adding the functions to generate detailed statistics of LI data cache access and enable the LI data cache model to selectively bypass identified data addresses. The detailed statistics results include information of data reuse time and load utilization rate of each memory access with different addresses, which are automatically analyzed by scripts to generate the list of bypassing addresses for each SM separately. The bypassing addresses are annotated and the benchmarks are simulated again with *GPGPU-Sim* [7] with the bypassing function enabled to implement the profiling-based cache bypassing method.

C. Hardware-based Bypassing

Based on the observations that the number of memory accesses of one load instruction distributes from 1 to 32 and that the distribution of the numbers of accesses per instruction is different among different kernels and benchmarks, we propose a hardware-based bypassing

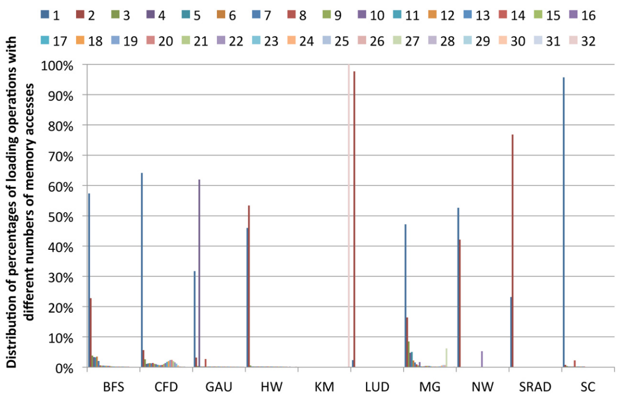


Fig. 5. Distribution of number of memory access.

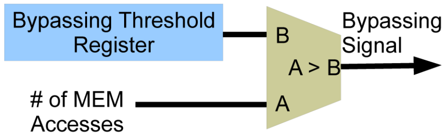


Fig. 6. Hardware-based bypassing scheme.

method, in which the load operations with a large number of memory accesses are bypassed, because the larger the number of accesses the worse the spatial locality. Fig. 5 shows the distribution of percentages of load operations with different numbers of memory accesses among the total number of load operations.

In this hardware-based bypassing scheme, we introduce a threshold register, as shown in Fig. 6, to store the bypassing threshold number of memory accesses per load instruction. For example, if the threshold register is set to the value of 16, at runtime, all the memory access, generated by the load instructions with the number of memory access larger than 16 per load operation, will bypass the Ll data cache.

D. Hybrid Selective Bypassing

In the hardware-based bypassing scheme, we bypass memory access from load operations that generate a relatively large number of memory access, because spatial locality is low in such cases. However, the temporal locality is not considered part of the hardware-based bypassing method. We also observe that some memory access from load operations with a large number of memory accesses may have good temporal locality, making it profitable to cache these memory accesses rather than to bypass them. Therefore, we propose a hybrid selective bypassing method, in which we collect the average number of reuse time and average utilization rate of the memory access in each memory access number distribution (1 to 32), and we use the same heuristic for the profiling-based bypassing method to evaluate whether a specific distribution should be bypassed.

Table 1. Hybrid Selective bypassing analysis of the benchmark *streamcluster*

Memory access number	Average reuse times	Average utilization rate	(R+1)*U
1	9.08	0.62	6.28
2	0.12	0.56	0.63
3	0.22	0.51	0.62
4	0.24	0.66	0.82
5	0.78	0.35	0.61
6	0.73	0.29	0.50
7	0.10	0.79	0.87
8	9.54	0.96	10.16
9	20.00	0.25	5.25
10	1.29	0.26	0.60
11	20.00	0.25	5.25
12	0	0.61	0.61
13	0	0.58	0.58
14 - 32	0	0	0

Table 1 shows the profiling information of a benchmark *streamcluster* (from the Rodinia benchmark suite [5]) needed by the hybrid selective bypassing method. In Table 1, the first column shows the distribution of memory access number (1 to 32); in this benchmark, there is no load operation with more than 13 memory accesses. The second and third columns show the average reuse times and utilization rate of the memory access in each distribution in the first column. The last column shows the result of (R+1) * U of each distribution. As shown in Table 1, the load operations with low spatial locality can have good temporal locality, such as those with 8, 9 and 11 memory accesses. Indicating that, in this benchmark, these load operations shall not be bypassed, while load operations with relatively better spatial locality but worse temporal locality, such as those with 6 memory accesses should be bypassed. In this hybrid selective bypassing method, to find the best bypassing threshold, the (R+1) * U value of each distribution is used as the bypassing threshold. For instance, if the (R+1) * U value of 0.62 is chosen as the threshold, the distribution with the value larger than 0.62 (1, 2, 3, 4, 7, 8, 9 and 11) will be cached, while the rest will be bypassed.

In the hybrid selective cache bypassing, the threshold in the bypassing threshold register is still the same as the hardware-based bypassing scheme, i.e., 16 memory accesses per load operation. The difference is that in the hybrid selective scheme, the profiling information is first used to annotate those that will not be bypassed, including the load operations that may have more than 16 memory accesses but are highly reused, so that U * (l + R) is

Table 2. Default *GPGPU-Sim* configuration

Number of SMs	15
Number of 32-bit registers per SM	32768
Size of L1 data cache per SM	16 kB
L1 data cache block size	128 B
L1 data cache associativity	4
Size of shared memory per SM	48 kB
Size of L2 cache	768 kB
DRAM latency cycles	100
Core clock frequency	700 MHz

larger than 1 (Eq. (1)). The load operations that are not annotated are candidates for cache bypassing at runtime by using the hardware-based bypassing.

V. EVALUATION METHODOLOGY

In this study, we use the *GPGPU-Sim* [7] simulator to implement and evaluate the proposed cache bypassing scheme. Table 2 shows the default configuration of the simulator, which simulates the Fermi GTX 480 platform. The benchmarks used in this study are from the *Rodima* [5] benchmark suite.

VI. EXPERIMENT RESULTS

A. Performance Results

Fig. 7 shows the normalized performance results of different benchmarks with different bypassing methods. The results are normalized to the performance of a benchmark simulated without using any bypassing method. As shown in the results, the profiling-based and hybrid selective bypassing methods can boost performance for all benchmarks, while the hardware-based bypassing method can boost performance of most of the benchmarks. This is because the hardware-based method uses a bypassing threshold only based on how many memory accesses one load operation generates, i.e., the spatial locality, while the other two methods also consider temporal locality.

As we can observe, the performance improvement of different cache bypassing schemes varies across different benchmarks due to different memory access characteristics. In general, hardware-based bypassing outperforms profiling-based scheme for most of the benchmarks, because hardware-based bypassing can exploit runtime information, which is more accurate than profiling-based information. However, one exception is the benchmark MG, for which hardware-based bypassing has performance worse than either profiling-based method or baseline without bypassing at all. As shown in Fig. 5, MG has a sizable number of loads whose memory accesses are more than 16 yet they still have temporal locality. Therefore, the hardware-based chooses to simply bypass all of them, which negatively impacts performance. In comparison, the profiling-based method and the hybrid selective method can consider the temporal locality based on the profiling information, which helps to boost performance of MG.

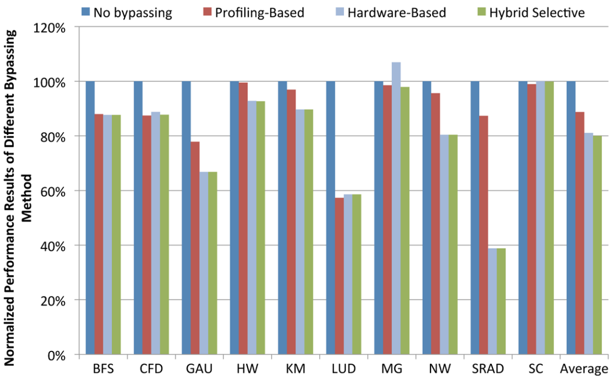


Fig. 7. Normalized performance results of different bypassing methods (the results of each bypassing method are normalized to the results without using bypassing).

B. Comparison with Compiler-Based Instruction Bypassing Scheme

Fig. 8 shows the normalized performance results of the Hybrid Selective bypassing method and the Compiler-Based bypassing method. Results reveal that two bypassing methods can achieve generally the same performance, while the improvement in the Compiler-Based method is slightly better. This is because, in the Compiler-Based method, the dependencies between different load instructions when the decision of bypassing one instruction is made, while this is not considered in the Hybrid Selective bypassing method.

However, both bypassing methods need to use profil-

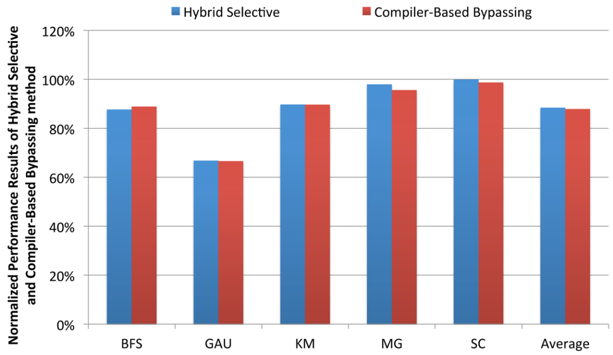


Fig. 8. Normalized performance results of Hybrid Selective and Compiler-Based bypassing methods (the results of each bypassing method are normalized to the results without using bypassing).

Table 3. Profiling costs of Hybrid Selective and Compiler-Based bypassing methods

	BFS	CFD	GAU	HW	KM	LUD	MG	NW	SRAD	SC	Average
Hybrid Selective	24	30	21	7	1	1	36	3	2	13	13.8
Compiler-based	38	949	25	1379	2	1319	77	381	101	46	431.7

ing information to generate the bypassing patterns. The Compiler-Based method uses profiling information to decide which load instruction(s) to bypass, while the Hybrid Selective method enables/disables load instruction(s) with a specific number of memory accesses (1–32). Table 3 shows the profiling costs of these two methods. The numbers in Table 3 indicate the times of execution of a benchmark/program to get the profiling information for each bypassing method. As the results reveal, the Hybrid Selective bypassing method needs much less profiling cost, while it can achieve performance improvement comparable to the Compiler-Based bypassing method.

VII. RELATED WORK

As GPUs are widely used in accelerating GPGPU applications, there have been an increasing number of studies to optimize GPU performance. Various compiler and runtime algorithms were proposed to improve the efficiency of GPU global memory [8-16] or shared memory [17, 18]. However, none of these studies targeted GPUs with caches.

GPU Cache Bypassing. Jia et al. [1] characterized application performance on GPUs with caches and proposed a compile-time algorithm to determine whether each load should use the cache. Their study first revealed that unlike CPU caches, the LI cache hit rates for GPUs did not correlate with performance. Recently, Xie et al. [2] studied a compiler-based algorithm to judiciously select global load instructions for cache access or bypass. Both Jia et al. and Xie et al.'s approaches can achieve performance improvement through cache bypassing. However, both approaches make cache bypassing decisions based on each global load instruction, which can access a variety of data addresses with diverse temporal and spatial locality. In contrast, our method is based on data addresses, not load instructions. This gives us finer-grained control on which data to be cached or bypassed to further enhance performance and energy efficiency.

Mekkat et al. [19] proposed Heterogeneous LLC (last-level cache) Management (HeLM), which can throttle GPU LLC access and yield LLC space to cache sensitive CPU applications. The HeLM takes advantage of the GPU's tolerance for long memory access latency to provide an increased share of the LLC to the CPU application for better performance. There are several major differences between HeLM [19] and our work. HeLM targets the shared LLCs in integrated CPU-GPU architectures, while our study focused on bypassing LI data caches in GPUs. Moreover, HeLM is a hardware-based approach that needs additional hardware extension to monitor thread-level parallelism (TLP) available in the GPU application. In contrast, our cache bypassing method is a software-based approach that leverages profiling information statically, which is simple and low cost and is particularly useful for embedded and mobile GPUs. Moreover, our method is complementary to the hardware-based HeLM, which can be used in conjunction with HeLM to further improve the GPU performance or energy efficiency in integrated CPU-GPU architecture.

Chen et al. [20] designed a hardware sampling based method on GPUs for LI data cache bypassing and used warp throttling to reduce contention. Tian et al. [21] implemented a PC-based dynamic GPU cache bypassing predictor. Xie et al. [22] recently studied a coordinated static and dynamic cache bypassing, in which a subset of thread blocks is analyzed at runtime to bypass the LI cache. Li et al. [23] proposed to use locality monitoring mechanism to dynamically bypass LI data caches for GPUs. Compared to all these studies that require non-trivial hardware support, our method is based on a considerably simpler hardware extension, which consists of a threshold register and a comparator. Also, our method can be easily integrated with the profiling-based method to further boost efficiency of the GPU LI data cache bypassing to enhance performance.

CPU Cache Bypassing. Cache bypassing has been extensively studied for CPUs in the past. Some architectures have introduced ISA support for cache bypassing, for example HP PA-RISC and Itanium. Both hardware-based [24-28] and compiler-assisted [29, 30] cache bypassing techniques have been proposed to reduce cache pollution and boost performance.

However, most CPU cache bypassing approaches use hit rates as performance metrics to guide cache bypassing, which may not be applicable to GPUs due to the distinct architectural characteristics and the non-correlation of GPU performance with data cache hit rates [1].

VIII. CONCLUSIONS

While GPUs have become popular in high performance computing, GPU applications exhibit different program behavior than traditional CPU programs. To support GPGPU applications with various data access patterns, recent GPUs such as NVIDIA Fermi and Kepler have introduced cache memory. While cache memories are generally useful to boost memory performance for CPUs, recent studies reveal that the cache hit rates are not correlated to GPU performance [1] due to the unique architectural features and program characteristics of GPUs. Therefore, it is critical to study how to exploit cache memory effectively for GPGPU applications. Specifically, due to the increasing use of GPUs in the embedded domain, it is crucial to develop a cost-effective method to use GPU caches efficiently for attaining better performance with less compilation, profiling, or optimization costs.

In this study, we examined a simple yet effective hardware-based method to bypass L1 data cache for GPGPU applications without recompiling programs. To exploit runtime and profiling information, we also explored a hybrid method, which can achieve better performance. Our evaluation indicates that the hardware-based cache bypassing boosts performance for most GPU benchmarks. The hybrid method can achieve performance comparable to state-of-the-art compiler-based bypassing approach [2], while reducing profiling and optimization costs significantly.

ACKNOWLEDGMENTS

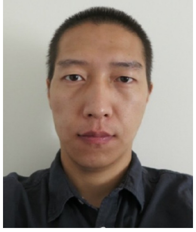
This study was funded in part by the NSF (Grant No. CNS 1421577).

REFERENCES

1. W. Jia, K. A. Shaw, and M. Martonosi, “Characterizing and improving the use of demand-fetched caches in GPUs,” in *Proceedings of the 26th ACM International Conference on Supercomputing*, Venice, Italy, 2012, pp. 15-24.
2. X. Xie, Y. Liang, G. Sun, and D. Chen, “An efficient compiler framework for cache bypassing on GPUs,” in *Proceedings of 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD),* San Jose, CA, 2013, pp. 516-523.
3. NVIDIA, CUDA Programming Guide version 5.5, https://developer.nvidia.com/cuda-toolkit-55-archive.
4. NVIDIA, “NVIDIA’s next generation CUDA compute architecture: Fermi,” 2009 [Internet], https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture _Whitepaper.pdf.
5. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, “Rodinia: a benchmark suite for heterogeneous computing,” in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC 2009)*, Austin, TX, 2009, pp. 44-54.
6. NVIDIA, Parallel Thread Execution ISA version 4.0, https://developer.nvidia.com.
7. A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009),* Boston, MA, 2009, pp. 163-174.
8. M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “A compiler framework for optimization of affine loop nests for GPGPUs,” in *Proceedings of the 22nd Annual International Conference on Supercomputing*, Island of Kos, Greece, 2008, pp. 225-234.
9. E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, “Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, Tsukuba, Japan, 2010, pp. 115-126.
10. Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU compiler for memory optimization and parallelism management,” *ACM SIGPLAN Notices*, vol. 45, no. 6, pp. 86-97, 2010.
11. I. J. Sung, J. A. Stratton, and W. M. W. Hwu, “Data layout transformation exploiting memory-level parallelism in structured grid many-core applications,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria,2010, pp. 513-522.
12. J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, “Many-thread aware prefetching mechanisms for GPGPU applications,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, 2010, pp. 213-224.
13. M. Rhu, M. Sullivan, J. Leng, and M. Erez, “A locality-aware memory hierarchy for energy-efficient GPU architectures,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, Davis, CA, 2013, pp. 86-98.
14. A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated scheduling and prefetching for GPGPUs,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 332-343, 2013.
15. M. Bauer, H. Cook, and B. Khailany, “CudaDMA: optimizing GPU memory bandwidth via warp specialization,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, 2011.
16. D. H. Woo and H. S. Lee, “COMPASS: a programmable data prefetcher using idle GPU shaders,” *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 297-310, 2010.
17. M. Moazeni, A. Bui, and M. Sarrafzadeh, “A memory optimization technique for software-managed scratchpad memory in GPUs,” in *Proceedings of IEEE 7th Symposium on Application Specific Processors (SASP'09),* San Francisco, CA, 2009, pp. 43-49.
18. Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, “Shared memory multiplexing: a novel way to improve GPGPU throughput,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, 2012, pp. 283-292.
19. V. Mekkat, A. Holey, P. C. Yew, and A. Zhai, “Managing shared last-level cache in a heterogeneous multicore processor,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Edinburgh, Scotland, 2013, pp. 225-234.
20. X. Chen, L. W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. M. Hwu, “Adaptive cache management for energy-efficient GPU computing,” in *Proceedings of the 47th Annual*

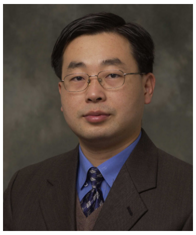
IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014, pp. 343-355.

21. Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jimenez, "Adaptive GPU cache bypassing," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, San Francisco, CA, 2015, pp. 25-35.
22. X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA),* Burlingame, CA, 2015, pp. 76-88.
23. C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, Newport Beach, CA, 2015, pp. 67-77.
24. A. Gonzalez, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proceedings of the 9th International Conference on Supercomputing*, Barcelona, Spain, 1995, pp. 338-347.
25. G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, 1995, pp. 93-103.
26. T. L. Johnson, D. A. Connors, M. C. Merten, and W. M. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338-1354, 1999.
27. H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: a new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of 2008 41st IEEE/ACM International Symposium on Microarchitecture (MICRO-41),* Lake Como, Italy, 2008, pp. 222-233.
28. M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433-447, 2008.
29. Y. Wu, R. Rakvic, L. L. Chen, C. C. Miao, G. Chrysos, and J. Fang, "Compiler managed micro-cache bypassing for high performance EPIC processors," in *Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35),* Istanbul, Turkey, 2002, pp. 134-145.
30. Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proceedings of 2002 International Conference on Parallel Architectures and Compilation Techniques,* Charlottesville, VA, 2002, pp. 199-208.



Yijie Huangfu

Yijie Huangfu received his B.S. degree in the major of Automation and M.S. degree of Communication and Information System from the Dalian University of Technology in 2006 and 2009, respectively. He received his Ph.D. degree in Electrical and Computer Engineering from the Virginia Commonwealth University in 2017. He has become an Architecture Engineer at NVIDIA since June 2017.



Wei Zhang

Wei Zhang is a professor in the Department of Electrical and Computer Engineering at Virginia Commonwealth University. Dr. Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, he worked as an assistant professor and then as an associate professor (tenured) at Southern Illinois University Carbondale (SIUC). His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. He has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. Dr. Zhang has received 5 research grants from the National Science Foundation. In addition, his research and educational efforts have been supported by industry including leading IT companies such as IBM, Intel, Motorola, and Altera. He has published more than 120 papers in refereed journals and conference proceedings. He is a senior member of the IEEE, and an associate editor of the Journal of Computing Science and Engineering. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.