Profiling-Based L1 Data Cache Bypassing to Improve GPU Performance and Energy Efficiency

Yijie Huangfu and Wei Zhang
Department of Electrical and Computer Engineering
Virginia Commonwealth University
{huangfuy2,wzhang4}@vcu.edu

***Abstract*—While caches have been studied extensively in the context of CPUs, it remains largely unknown how to exploit caches efficiently to benefit GPGPU programs due to the distinct characteristics of CPU and GPU architectures. In this work, we analyze the memory access patterns of GPGPU applications and propose a cost-effective profiling-based method to identify the data accesses that should bypass the L1 data cache to improve performance and energy efficiency. The experimental results show that the proposed method can improve the performance by 13.8% and reduce the energy consumption by about 6% on average.**

I. Introduction

Graphics Processing Units (GPUs), originally designed for fast graphical computation, have rapidly become a popular choice for high-performance computing. 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 a C-like programming language to write general-purpose code for execution on GPUs. The tremendous computation power and the enhanced programmability of GPUs greatly accelerate the general-purpose computing on GPUs (GPGPUs) to achieve superior performance or energy efficiency.

In the past few years, GPUs have been increasingly used in high-performance embedded computing. The massive parallelism and energy efficiency of GPUs can benefit a wide variety of data-parallel embedded applications including imaging, audio, video, military, and medical applications [1]. Actually, there are already several embedded GPUs that can be integrated on System-on-Chips (SoCs) for mobile devices, for example ARM's Mali graphics processor, Vivante's embedded graphics core, and the StemCell Processor from ZiiLabs (Creative).

Lately, major GPU vendors have introduced cache memories in conjunction with the shared memory to benefit a wide variety of GPGPU applications. For example, both the 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 the share memory are configurable while the aggregate on-chip memory size is fixed. Although the cache memory can effectively hide the access latency for data with good temporal and/or spatial locality for both CPUs and GPUs, GPGPU applications may exhibit divergent memory access patterns from traditional CPU applications. Moreover, the recent study shows that GPU caches have counter-intuitive performance tradeoffs [4]. Therefore, it is important to explore techniques to exploit the on-chip cache memories effectively to boost GPU performance. In particular, for embedded and mobile GPU applications, it is also crucial to develop cost-effective optimization methods for improving performance and energy efficiency.

In this paper, we examine the memory access characteristics of GPGPU applications based on the Rodinia benchmarks [7]. We find that unlike CPU applications that generally exhibit good temporal or spatial locality, a large fraction of memory accesses in these GPGPU applications are not reused at all or just reused for a few number of times. Also, GPGPU applications have diverse utilization rates for data loaded from the global memory. To reduce the pressure on GPU global memory bandwidth and to decrease GPU cache pollution, we propose a profiling based method to bypass those data accesses for improving performance and energy efficiency.

II. Motivation

A. Global Load Utilization Rate

A GPU typically consists of an array of highly threaded streaming multiprocessors(SMs), and each SM has a number of Streaming Processors (SPs) that can execute threads in parallel. When a GPU kernel is launched, the runtime creates massive concurrent GPU threads organized hierarchically. A number of 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 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 Figure 1 (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 actually used by the GPU, is 100% (128/128). However, when the memory access pattern changes a little bit, as shown in Figure 1 (b) and (c), the address range becomes 96 to 223, which spans across the

Copyright is retained by the authors

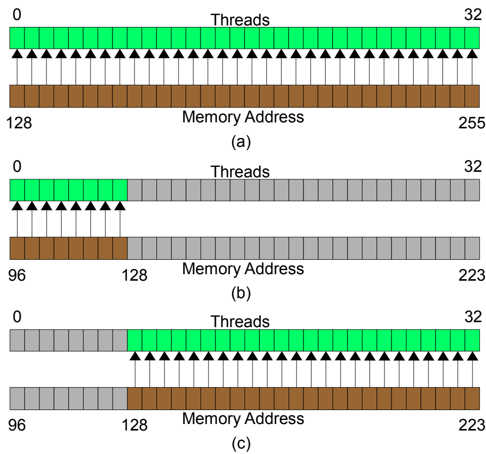
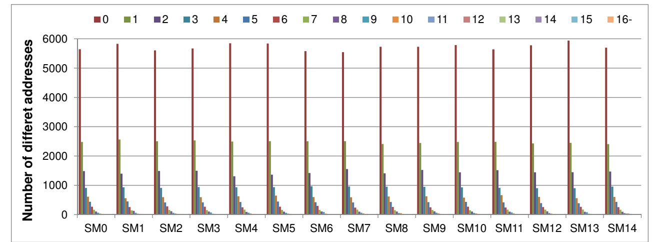
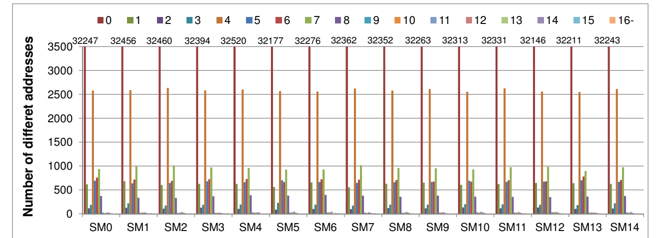


Fig. 1. Examples of different memory access patterns that lead to various global load utilization rates.



(a) Data reuse distribution of *gaussian* benchmark



(b) Data reuse distribution of *srad* benchmark

Fig. 2. The data reuse distribution in the L1 data cache.

boundary of 128 bytes. In this case, two 128-byte transactions are needed to transfer the data needed by the threads. Thus the 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 degrade both performance and energy efficiency for GPGPU computing.

Moreover, in the CUDA programming model, if the required data are cached in both the L1 and L2 data caches, memory accesses are done by 128-byte transactions. However, if the data are only stored into the L2 cache (i.e. bypassing the L1 data cache), 32-byte transactions are used instead [3]. Therefore, for the load operations depicted in Figure 1 (b) and (c), assuming the data are not reused from the cache, using only 32-byte transactions can reduce the over-fetching of useless data and therefore decrease the memory traffic.

B. Data Reuse Times in GPU L1 Data Cache

The GPGPU applications usually operate on massive amount of data. However, the cache line usage among the data with different addresses may differ significantly. This is not only because GPGPU applications can exhibit irregular data access patterns, but also because the effective L1 data cache space per SP is too small. Thus even if some data are reused within a warp, they 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 accesses.

Figure 2 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 [7]. The experimental configuration and the evaluation methodology are detailed in Section IV. 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 see, 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 number of data addresses are never reused at all or are only reused for a very small number of times. For example, in *gaussian*, nearly half of the addresses are used for just once, while in the *srad* the majority of the addresses are not reused at all. The very low temporal locality from GPGPU applications is quite different from typical CPU applications that tend to have good temporal locality; therefore, we need to explore novel cache management techniques for GPUs.

For data that are never reused at all, loading them into the cache is not helpful to reduce neither latency nor memory bandwidth. On 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 the global memory traffic if the load utilization rate is low. This may negate the benefit of a small number of cache hits. Therefore, it becomes attractive to bypass those data that are never reused or only reused a few times to reduce the memory bandwidth pressure and cache pollution for GPUs.

III. GPU L1 Data Cache Bypassing

A. Bypassing by Addresses vs. by Instructions

By default, on the CUDA platform, global memory accesses are cached in both the 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]. Based on this mechanism, Xie et al. recently proposed a compiler-based GPU cache bypassing framework to improve performance [5].

While Xie et al's approach [5] can automatically analyze the GPU code and select global load instructions for cache access or bypassing intelligently, it relies on using Integer Linear

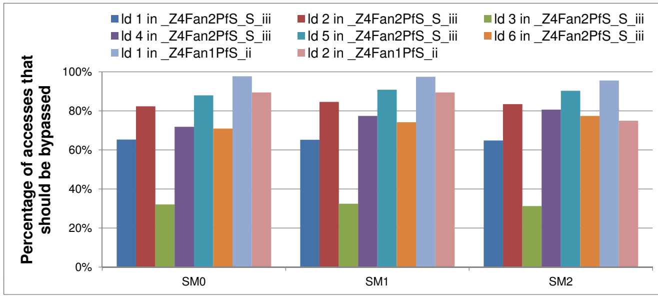


Fig. 3. Percentage of accesses that should be bypassed of each load instruction in the kernels of *gaussian* benchmark.

Programming (ILP) to solve the traffic reduction maximization problem exactly, which unfortunately is not scalable to large problems. Even with a heuristic based approach, the algorithm is iterative, and only one node in the traffic reduction graph can be analyzed for bypassing or not in each iteration, which can still take significant time for determining cache bypass or not for large applications.

Also, making cache bypassing decisions solely based on each global load instruction not be very effective for applications whose cache access patterns exhibit large variation. This is because each global load instruction can access a wide range of data addresses, which can have distinct reuse behaviors. Bypassing at the load instruction level is only effective if the data accessed by the load instruction have uniform locality. For instructions that access data with diverse reuse characteristics, making bypassing decisions at the instruction level is too coarse-grained, i.e. it can only choose between bypassing or caching all the data accessed by this instruction, not according to the subset of data accesses 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.

Figure 3 shows, according to each load instruction of the benchmark *gaussian*, the percentage of the memory accesses to different addresses that have low L1 cache line reuse time and low load utilization rate (see Equation 1 in Section III-B for details). We focus on analyzing 8 load instructions in 2 kernels (ld 1-6 of kernel *_Z4Fan2PfS_S_iii* and ld 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 ld1 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 accesses from one load instruction will cause performance overhead for those data accesses that should not be bypassed. Similarly, simply caching all accesses from one load instruction will lose the performance improvement opportunity for those data accesses that should be bypassed. To facilitate finer-grained control of cache bypassing, in this work, we choose 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 the L1 data cache accesses that should be bypassed. We focus on bypassing the data accesses that have low load utilization rates and low reuse times in the L1 data cache, with the objective to minimize the 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 Equation 1 to determine which data accesses should be bypassed.

$$U \times (1 + R) < 1 \tag{1}$$

In the above equation, $(1 + R)$ represents the number of times A is accessed from the L1 data cache, including the first time when it is loaded into the cache, i.e., 128 bytes are transferred from the 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 Equation 1 holds, then storing A into the L1 data cache will actually increase the global memory traffic as compared to bypassing this access from the L1 data cache. Therefore, in this case, bypassing A can reduce the 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 L1 data cache.

Our cache bypassing method considers both 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 Figure 1 (b), i.e., $U = 25\%$, this address must be reused at least 3 times in the L1 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 Figure 1 (c), i.e., $U = 75\%$, if this address is reused at least once from the L1 data cache (i.e., $R \geq 1$), then it should not be bypassed.

IV. Evaluation

In this work, we use the *GPGPU-Sim* [8] simulator to implement and evaluate the proposed cache bypassing scheme. Table I shows the default configuration of the simulator, which simulates the Fermi GTX 480 platform. The benchmarks used in this work are from the *Rodinia*[7] benchmark suite.

TABLE I. Default *GPGPU-Sim* Configuration

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

Figure 4 shows the performance improvement of the proposed cache bypassing method, which is normalized to the total number of execution cycles without bypassing. As we can see, the cache bypassing method improves the performance for

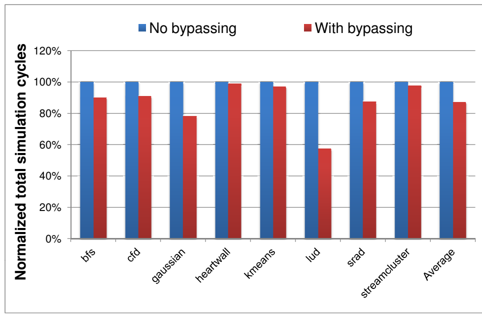


Fig. 4. Numbers of execution cycles with and without cache bypassing, normalized to that without cache bypassing.

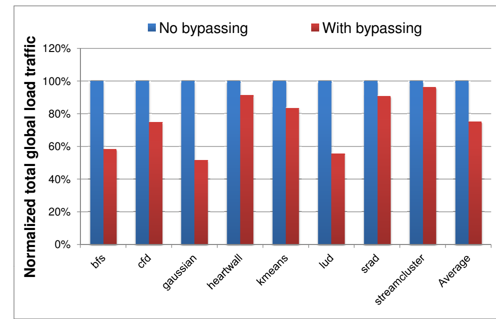


Fig. 5. Normalized global memory traffic with and without cache bypassing, which is normalized to that without cache bypassing.

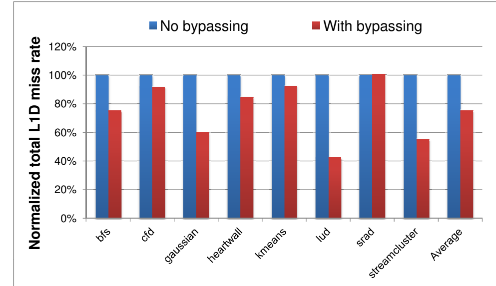


Fig. 6. Normalized L1 data cache miss rates with and without cache bypassing, which are normalized to that without cache bypassing.

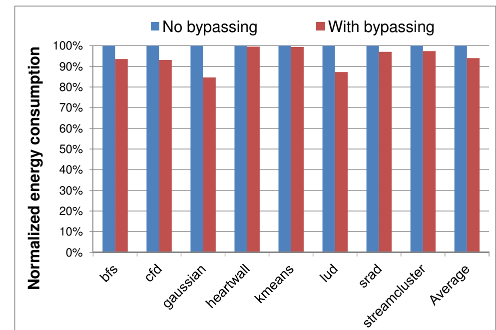


Fig. 7. Normalized energy consumption results with/without bypassing.

all benchmarks. The total number of execution cycles for *lud* is reduced by more than 40%, and the average reduction of execution cycles for all benchmarks is 13.8%.

The performance improvements come from two factors. The first factor is the reduction of the global memory traffic caused by cache bypassing, which is shown in Figure 5. The results indicate that the global load memory traffic can be reduced by 24.7% on average among the benchmarks simulated in this work. The second factor of performance improvement is the reduction of L1 data cache miss rates as shown in Figure 6. We observe the cache miss rate is reduced by up to 57.5% for *lud*, and the average reduction is 24.6%.

Particularly, we find that when cache bypassing reduce both global memory traffic and cache miss rates, the performance is improve dramatically. For example, for both *lud* and *gaussian*, both the global memory traffic and cache miss rates are reduced significantly. As a result, the performance of *lud* and *gaussian* is improved by 42.7% and 21.8%. In contrast, for some benchmarks such as *streamcluster*, although cache bypassing reduces its cache miss rate by 44.8%, its global memory traffic is only reduced by 3.8%, leading to small performance improvement of 3.4%. This also indicates that reducing memory traffic may be more important than reducing cache miss rates for GPGPU programs.

It should also be noted that the proposed bypassing method does not necessarily reduce the L1 data cache miss rate, for example *srad*, because the total number of accesses to the L1 data cache is also reduced by cache bypassing. However, on average, the L1 data cache miss rate is reduced by 24.6%, indicating that the proposed cache bypassing method can effectively alleviate cache pollution and thus improve performance.

We use the energy model *GPUWattch* [9], which is integrated with *GPGPU-Sim* [8], to evaluate the effect of the proposed cache bypassing method on the total energy consumption of the simulated GPU platform. Figure 7 shows the normalized energy consumption results of the simulated GPU platform with and without the L1 data cache bypassing. The results indicate that the energy consumption can be reduced by about 6% on average. The energy saving comes from the reduction of both the global memory traffic and the total number of execution cycles.

V. Conclusion

In this work, we study a simple yet effective method to profitably bypass the L1 data cache for GPGPU applications. Our method exploits the profiling information to identify the global memory load accesses with low utilization rates and low L1 data cache reuse times, enabling the GPU to access these addresses directly from the L2 cache while bypassing the L1 data cache. Our experiment results show that the proposed bypassing method can effectively reduce the global load memory traffic and the number of L1 data cache misses, and hence improve the performance of GPGPU applications. On average, the proposed method reduces the global memory traffic by 22.2%, improves the performance by 13.8%, and reduces energy consumption by about 6%.

Acknowledgment

This work was funded in part by the NSF grant CNS-1421577.

References

[1] J. D. Owens, et al. GPU computing. Proceedings of the IEEE, 2008.

[2] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[3] NVIDIA Corp. CUDA Programming Guide, Version 5.5.

[4] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In ICS, 2012.

[5] X. Xie, Y. Liang, G. Sun and D. Chen. An efficient compiler framework for cache bypassing on GPUs. In Proc. of IEEE/ACM International Conference Computer-Aided Design, 2013.

[6] V. Mekkat, A. Holey, P. Yew and A. Zhai. Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor. In Proc. of PACT, 2013.

[7] S. Che, et al. Rodinia: A benchmark suite for heterogeneous computing. In Proc. of the IEEE Int. Symp. Workload Characterization, 2009.

[8] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.

[9] J. Leng et al. GPUWattch: enabling energy optimizations in GPGPUs. In Proc. ISCA, 2013.