A Survey of Techniques for Managing and Leveraging Caches in GPUs

Sparsh Mittal

Journal of Circuits, Systems, and Computers

A Survey of Techniques for Managing and Leveraging Caches in GPUs

SPARSH MITTAL

Future Technologies Group, Oak Ridge National Laboratory (ORNL)
Oak Ridge, Tennessee, 37830 United States.
mittals@ornl.gov

Initially introduced as special-purpose accelerators for graphics applications, GPUs have now emerged as general purpose computing platforms for a wide range of applications. To address the requirements of these applications, modern GPUs include sizable hardware-managed caches. However, several factors, such as unique architecture of GPU, rise of CPU-GPU heterogeneous computing etc., demand effective management of caches to achieve high performance and energy efficiency. Recently, several techniques have been proposed for this purpose. In this paper, we survey several architectural and system-level techniques proposed for managing and leveraging GPU caches. We also discuss the importance and challenges of cache management in GPUs. The aim of this paper is to provide the readers insights into cache management techniques for GPUs and motivate them to propose even better techniques for leveraging the full potential of caches in the GPUs of tomorrow[a].

Keywords: GPU (graphics processing unit); GPGPU (general purpose GPU); cache memory; performance; energy efficiency; classification

1. Introduction

Recent years have witnessed a phenomenal growth in the capabilities and applications of graphics processing units (GPUs). GPUs, which were initially introduced as special-purpose accelerators for games and graphics code are now used as ubiquitous high-performance computing platforms, in systems ranging from hand-held embedded systems to massive supercomputers. This has led to the emergence of GPGPU (general-purpose GPUs) field.

The demands of new application domains have motivated novel changes in GPU[b] design and architecture. Traditionally GPUs only provided software-managed local memories, however as the application domain of GPU broadens, these memories become insufficient in fulfilling the need of applications running on GPUs. To address this challenge, state-of-the-art GPUs provide hardware-managed multi-level

[a]Preprint version of paper accepted in Journal of Circuits, Systems and Computers (JCSC) World Scientific, article number 1430002, to be published in Vol. 23, No.8 (September 2014). DOI: 10.1142/S0218126614300025 URL http://www.worldscientific.com/doi/abs/10.1142/S0218126614300025
[b]In the rest of the paper, we use the term GPU to also refer to a GPGPU.

caches. In fact, introduction of caches in Fermi has been seen as one of the "Top 10 most important innovations" in the GPU architecture" [1], which has helped GPUs to move towards mainstream computing. Caches affect application performance in a significant manner, as confirmed by several researchers [2,3,4,5,6,7,8,9,10,11]. This makes management of GPU caches extremely important. While CPU cache management has been studied over years, GPU cache management is a relatively new research field [12,13,14].

In this paper, we present a survey of techniques for managing and leveraging GPU caches. Further, we present a classification of these on the basis of their key characteristics. We also discuss the factors which motivate the importance of GPU cache management. Since it is not possible to cover the full range of research works in the area of GPU cache management, we take the following approach to restrict the scope of the paper. We only discuss techniques proposed for managing GPU caches, although some techniques proposed in the context of CPU caches may also be beneficial for GPU caches. We mainly discuss hardware-managed caches and not software-managed caches. We focus on architectural and system-level techniques and not device-level (VLSI-design level) techniques. We present the key idea of each research work and do not present the quantitative results since different techniques have been evaluated on different platforms. We discuss techniques dealing with issues such as performance and energy efficiency and not dealing with other aspects such as reliability. The objective of this paper is to equip the researchers, application-developers and architects with the knowledge of importance and working of GPU cache management techniques and motivate them to propose novel solutions for architecting GPU caches of tomorrow.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of GPU architecture and its evolution over time. Section 3 summarizes the importance of cache management techniques in GPUs. Section 4 discusses several cache management techniques in detail. Finally Section 5 presents the conclusion.

2. GPU Architecture and State-of-the-art GPUs

In this section, we briefly review the GPU architecture, as relevant to this paper and refer the reader to prior work for more details [15]. GPUs have been designed to provide very high computational power and are suited for applications which are throughput-oriented and not latency-sensitive. GPUs use numerous programmable computational cores and fine-grained threads. Moreover, to achieve scalability, GPUs minimize use of global structures. For example, unlike CPUs, the streaming-multiprocessors (SMs) in GPUs have simple in-order pipelines. GPU chips spend more die-space on ALUs and less on caches, thus instead of seeking performance via out-of-order processing over large instruction windows and large caches, they incorporate low-overhead thread scheduling and hide memory latencies via multithreading. Further, to exploit data access locality, GPUs feature large register files, shared memory and relatively small caches. For example, Intel's Itanium

9560 CPU uses 32MB last level cache (LLCs) [16]. In contrast, the GT200 architecture GPUs did not feature an L2 cache, the Fermi GPU has 768KB LLC and the Kepler GPU has 1536KB LLC [17].

3. Importance of Techniques for Managing and Leveraging GPU Caches

In this section, we first discuss the factors that affect the efficiency of GPU caches to highlight the limitations of the existing cache management techniques. We then discuss the motivations and challenges for designing novel techniques for GPU cache management.

3.1. *Limitations of existing cache management techniques*

The need to account for unique GPU characteristics: Conventional cache management techniques (e.g. LRU replacement policy, coherence protocols etc.) have been designed to exploit the architectural characteristics of CPU and application behavior of serial applications. However, GPU architecture and programming are significantly different from their CPU counterpart. This fact introduces several new constraints and performance-objectives and hence, the effectiveness of conventional cache management techniques is greatly reduced when they are used in GPUs [18,19,20]. Also, demand-fetched caches are only recent additions in GPUs and hence, designers have very little intuition about which applications will benefit from these caches.

The need to cope with small cache size: GPU caches are shared by thousands of active threads which makes the cache a scarce resource. This may lead to cache contention in the case when the required dataset size of thread groups assigned to a SM cannot fit into the local cache [21,22,23]. In absence of caches, the bandwidth utilization of GPUs becomes very high, which significantly harms the DRAM performance due to queuing effects and memory access bursts [24].

The need to avoid negative effect of caches on performance: For several applications, caches may actually harm the performance [25,26]. For example, when cache line size exceeds the minimum fetch size of main memory, unnecessary data are fetched. Similarly, with write-allocate policy, data are copied to cache on a cache miss, although they are not reused for applications with high miss-rate. Moreover, given the limitations posed by chip-area, increasing the GPU cache size reduces the area available for ALUs, which also reduces the throughput. Thus, the GPU performance cannot be improved simply by increasing the cache sizes.

3.2. *Motivations and challenges for designing novel cache management techniques*

Achieving a balance between multithreading and cache usage efficiency: GPUs employ deep multithreading by which the control switches among numerous

active threads to hide the memory latency. This helps in achieving maximum level of execution parallelism. However, due to deep multithreading, different thread groups may replace the useful blocks of other threads (called interference), which may lead to poor performance [27,28]. Use of smaller number of threads alleviates this issue, however, this comes at the cost of reduced execution parallelism [29]. Thus, due to availability of multithreading in GPUs, an improvement in cache performance does not directly translate into improved program performance [30] and hence, intelligent cache management techniques are required for achieving a balance between the two factors.

Avoiding off-chip accesses and increasing bandwidth: To provide high performance, GPUs demand very high memory bandwidth. Due to the power-wall problem and the physical limitations of chip-packaging, achieving high bandwidth by increasing clock-frequency or pin-count is extremely challenging. Intelligent cache management policies can be highly useful in such scenarios since they can help in reducing the off-chip accesses by capturing data locality [31,32,33,20,26]. This also increases the effective memory bandwidth, which translates into improved performance and energy-efficiency. Several researchers have compared the performance of Tesla and Fermi GPUs and have observed that hardware-managed caches play an important role in offering Fermi GPUs a performance advantage over previous generation GPUs [34,35,36,37,38,39]. L2 caches in Fermi also improve the performance of atomic operations [40].

Managing shared cache in CPU-GPU heterogeneous computing processors: Since CPUs and GPUs are more suitable for different classes of applications, chip-designers have recently focused on CPU-GPU heterogeneous computing to achieve the best of two worlds. Such chips may feature a last level cache which is shared by both CPU and GPU, for example, on Intel's Sandy Bridge processor, CPU and GPU are on the same chip with a shared on-chip 8MB L3 cache [41]. It is well known that CPU and GPU applications have different characteristics and cache requirements, for example, GPUs have very large number of threads and hence, they may access the cache much more frequently, which would lead to starvation of CPU application. Hence, intelligent techniques are extremely important for managing caches in such heterogeneous computing systems.

4. GPU Cache Management Techniques

Table 1 presents a classification of the techniques proposed for GPU caches. In the remainder of the section, we summarize the key ideas of several of these techniques.

4.1. *GPU Memory Hierarchy Design/Exploration*

Although the primary function of the shared caches in GPGPUs is to reduce off-chip memory traffic rather than to hide memory latency, for applications with limited or no parallelism, caches can be highly useful for hiding the memory latency [57].

Table 1. Classification of Approaches For Managing and Leveraging GPU Caches

Classification	References
Application Domain	
For CPU-GPU heterogeneous systems	18,42,43,44
For discrete GPU systems	almost all others
Essential approach used for cache management	
Memory hierarchy redesign/exploration	45,46,13,47,8,48
Thread/warp scheduling	49,22,50,51,52,28,53,6
3d stacking and use of non-volatile memory	54
Power-gating (leakage control)	55
Cache partitioning	18,42
Cache bypassing	18,19
Prefetching	43,51
Improving cache hit-rate and avoiding interference	56,21,22,28,6
Goal of the cache management technique	
Implementing cache coherence	44,20
Improving performance	46,28,6
Saving energy	46,55,47,13

Gebhart et al. [46] present the design of a unified local memory in GPU which can dynamically change the partitioning among registers, cache, and scratchpad on a per-application basis. The existing designs use rigid partition sizes, however, different GPU workloads have different requirements of registers, caches and scratchpad (also called shared memory). Based on the characterization study of different workloads, they observe that different applications and kernels have different requirements of cache, shared memory etc. To address this, they propose a unified memory architecture that aggregates these three types of storage and allows for a flexible allocation on a per-kernel basis. Before the launch of each kernel, the system reconfigures the memory banks to change the memory partitioning. By virtue of effective use of local-storage, their design reduces the accesses to main memory. They have shown that using their approach broadens the range of applications that can be efficiently executed on GPUs and also provides improved performance and energy efficiency.

Sankaranarayanan et al. [45] propose adding small sized caches (termed as tinyCaches) between each lane in a streaming multiprocessor (SM) and the L1 data cache which is shared by all the lanes in an SM. Further, using some unique features of CUDA/OpenCL programming model, these tinyCaches avoid the need of complex coherence schemes and thus, afford low-cost implementation. They have shown that these small caches effectively filter a large fraction of memory requests that would otherwise need to be serviced by the first level cache or scratchpad memory. This leads to improvement in the energy efficiency of the GPU.

Lashgar et al. [47] propose a technique to reduce accesses to instruction cache and save energy by using filter-cache. Their technique aims to exploit "inter-warp instruction temporal locality" which means that during short execution intervals, a small number of static instructions account for a significant portion of dynamic

instructions fetched and decoded within the same stream multiprocessor. Due to this, the probability that a recently fetched instruction will be fetched again becomes high. They propose using a small filter-cache to cache these instructions, which reduces the number of accesses to instruction cache and improves the energy efficiency of the fetch engine. Filter-cache has been used in the context of CPUs also, however, in GPUs the instruction temporal locality is even higher. This is due to the fact that GPUs interleave thousands of threads per core, which are grouped in warps. The warp scheduler continuously issues instructions from different warps which fills the warp, thus it fetches the same instruction for all warps during short intervals.

Hughes et al. [13] study the performance and power implications of GPU caches. They study different last level cache designs, such as private, shared, caches with or without replications etc. They observe that shared LLC provides the best performance, while the private LLC provides the highest energy efficiency. This is because, the shared LLC provides highest throughput, while the private LLC minimizes on-die traffic.

Jia et al. [25] characterize GPU application performance on a real GPU with L1 caches turned on and off. They study the degree to which L1 caches may either improve or hurt program performance. In NVIDIA GPUs, L1 caches are not coherent across SMs, and hence, global memory writes (stores) ignore L1 caches. Hence, applications with global memory writes do not benefit from L1 caches. Those applications which use global memory (rather than shared memory) to hold their working sets achieve benefit from the use of L1 caches. Based on their observations, they provide a taxonomy of GPU memory access locality to systematically analyze the reasons about when caches are likely to be helpful. They also propose methods to enable automated compile-time optimizations to determine when to use/disable L1 caches in GPUs.

Ristov et al. [8] study the impact of different sizes and associativities of L1 and L2 caches on a GPU on the performance of matrix multiplication application. They observe that only L2 cache impacts the overall performance of the algorithm. Different configurations of L1 cache have only small effect on the performance of matrix multiplication algorithm.

Maashari et al. [54] study the impact of 3D stacking of caches (e.g. texture unit caches and Z caches) on GPU performance. They observe that compared to an iso-cost 2D GPU design, a 3D GPU design offers significant performance advantage. They also investigate use of non-volatile magnetic RAM (MRAM) for designing caches. Note that compared to SRAM, non-volatile memories have higher density and negligible amount of leakage energy, but they also have small write endurance and high write latency and energy [58]. They observe that due to high write latency of MRAM compared to SRAM, MRAM does not always provide performance advantage over SRAM, although use of MRAM is beneficial for improving energy efficiency.

4.2. *Microarchitectural Cache Management Issues/Policies*

Singh et al. [20] propose a time-based coherence framework for GPUs, that uses globally synchronized counters in a single-chip system to develop a streamlined GPU coherence protocol. GPUs lack cache coherence and if an application requires memory operations to be visible across all cores, the private caches must be disabled. Further, conventional cache coherence protocols introduce unnecessary coherence traffic overheads in GPUs and require very high amount of storage for tracking thousands of in-flight coherence requests. To eliminate the coherence traffic and avoid protocol races, Singh et al. use synchronized counters which enable all coherence transitions to happen synchronously. Their coherence framework works on the intuition that if the lifetime of a memory address' current epoch can be predicted and shared among all readers when the location is read, then the readers can leverage the counters to self-invalidate synchronously, eliminating the need for end-of-epoch invalidation messages.

Power et al. [44] present a framework for supporting directory-based hardware coherence between CPUs and GPUs in a heterogeneous CPU-GPU system. They assume a heterogeneous system where CPU and GPU clusters have two separate, non-inclusive, shared L2 caches. Their coherence scheme replaces a standard directory with a region directory and adds region buffers to L2 caches of both CPU and GPU to track the regions over which the CPU or GPU currently hold permission. These structures allow the system to move the coherence-related traffic from the coherence network to the high-bandwidth direct-access bus while still maintaining coherence.

Choi et al. [19] propose two cache management schemes for GPUs, viz. write-buffering and read-bypassing. Their schemes work by controlling the placement of data in the shared L2 cache to maximally reduce the memory traffic. By analysis of the code, data usage characteristics are identified, which is used to direct data placement of individual load or store instruction in the cache. With this support, the write-buffering technique utilizes the shared cache for inter-block communication to reduce memory traffic. The read-bypassing scheme attempts to avoid placing streaming data in the shared cache, that are consumed only within a thread-block. They have shown that their techniques significantly reduce the off-chip memory accesses.

Meng et al. [56] propose a technique to reduce conflict misses in LLC of GPUs. They note that the private data of each thread, which need not reside in the LLC, is one of the most important sources of thrashing in LLC. To reduce LLC conflicts and mitigate cache thrashing, they propose a run-time stack allocation mechanism that randomizes the offset of the stack bases relative to page boundaries. This leads to more uniform distribution of thread-private data in the LLC, which reduces conflict misses. They also study the effectiveness of different cache replacement policies in addressing this issue. Further, they propose a non-inclusive semi-coherent cache design which allows the private data to exist only in L1 cache.

Rhu et al. [59] propose a locality-aware technique for finding the right fetch granularity for improving performance and energy-efficiency of GPUs. Their approach enables adaptively adjusting the memory access granularity depending on the spatial locality present in the application. They show that only few applications use all the four 32B sectors of the 128B cache-block. This leads to over-fetching of data from the memory. To address this, they first decide the appropriate granularity (coarse-grain or fine-grain) of data fetch. Using this, a hardware predictor adaptively adjusts the memory access granularity without programmer or runtime system intervention.

4.3. *Thread Scheduling Policies*

Yen et al. [49] propose a hardware-based thread scheduler to dynamically adjust the degree of multithreading in GPU with the awareness of cache contention. Their technique works in two phases. In the first phase, called training phase, the statistics are collected from the L1 and L2 cache. In the second phase, called tuning phase, PID (Proportional Integral Derivative) control is used to dynamically adjust the degree of multithreading based on the information obtained from the first phase. Thus, when the system is short of cache resources, the degree of multithreading is reduced and in the case of low cache contention, degree of multithreading is increased to benefit from the massive parallelism.

Rogers et al. [22] propose a cache-conscious wavefront (warp) scheduling technique. Their technique uses a lost intra-wavefront locality[c] detector (LLD) which informs the scheduler if its decisions are destroying intra-wavefront locality. Based on this feedback, the scheduler assigns intra-wavefront locality scores to each wavefront and ensures that those wavefronts losing intra-wavefront locality are given more exclusive access to the L1 cache. Their technique effectively changes the re-reference interval to reduce the number of interfering references between repeated accesses to the high locality data, which reduces the thrashing in L1 cache.

Rogers et al. [28] propose a divergence-aware warp scheduling technique. Their technique uses a divergence-based cache footprint predictor to estimate how much L1 data cache capacity is needed to capture intra-warp locality in loops. These estimates are obtained from runtime information about the level of control flow divergence in warps and online characterization of memory divergence. Based on these estimates, warp scheduling is done in a manner that the data reused by active threads may not exceed the capacity of the L1 data cache. Thus, their technique minimizes interference in L1 cache.

Jog et al. [50] present a coordinated CTA (cooperative thread array[d]) aware

[c]Intra-wavefront locality is termed as the locality that occurs when data is initially referenced and re-referenced from the same wavefront.

[d]GPU applications are generally divided into several kernels, where each kernel spawns many threads. These threads are grouped together into thread blocks, which are known as cooperative thread arrays (CTAs). At the beginning of execution of an application, the CTA scheduler initiates

scheduling policy that aims to minimize the impact of long memory latencies. They propose a CTA-aware two-level warp scheduler that groups all the available CTAs on a core into smaller groups and schedules all the *groups* (instead of individual CTAs or warps) in a round-robin fashion. This allows a smaller group of warps/threads to access the L1 cache in a particular interval of time, which reduces the cache contention. It also reduces the inactive periods since the time at which different warps reach long latency operations becomes different and hence memory stalls can be effectively hidden. To further improve the L1 hit rate, they propose a locality-aware scheduling scheme, which always prioritizes a group of CTAs in a core over the rest of the CTAs until they finish. This helps in taking advantage of the locality between nearby threads and warps which are associated with the same CTA. This is because the higher priority CTAs can keep their data in private caches and get opportunity to reuse it.

Jog et al. [51] propose a prefetch-aware warp scheduling policy to effectively hide long memory latencies. Since consecutive warps are highly likely to access nearby cache blocks, prefetches are generated by a warp very close to the time their corresponding addresses are actually demanded by another warp. To address this, their technique separates in time the scheduling of consecutive warps such that they are not executed back-to-back. Thus, at the time when one warp stalls and generates its demand requests, a prefetcher can issue prefetches for the next N cache blocks, which are likely to be completed by the time the consecutive warps that use those blocks are scheduled. This improves the effectiveness of prefetching and also improves the L1 cache hit rate.

Meng et al. [53] propose a technique which allows dynamic sub-division of warps for hiding latency of branch and memory divergence. In a conventional SIMD (Single instruction, multiple data) implementation, branch or memory divergence stalls an entire warp. In their technique, upon a branch divergence, a warp can be divided into two active warp-splits, each representing threads that fall into one of the branch paths. The execution of these warp-splits is then interleaved. Similarly, when threads from a single warp experience different memory-reference latencies caused by cache misses, memory latency divergence occurs. In such cases, a warp is divided into two warp-splits, one with the threads whose memory operations have completed, the other represents threads that are still stalled on cache miss. The former warp-split runs ahead and can potentially prefetch cache lines that may also be needed by threads that fell behind. Upon future memory divergence, warp-splits can be recursively divided. They also provision methods to prevent over-subdivision since it has a negative effect on performance.

Guz et al. [52] present an analytical model to quantify the harmful effect of increasing the number of threads sharing the cache. They demonstrate that increasing the thread count improves performance until the total working set no longer fits

scheduling of CTAs onto the available cores. All the threads within a CTA are executed on the same cores.

in cache. Beyond this point, an increase in the number of threads degrades performance until enough threads are present to hide the system's memory latency.

4.4. *Cache Management Policies For CPU-GPU Heterogeneous Systems*

Lee et al. [18] propose a thread-level parallelism (TLP) aware cache management policy of CPU-GPU heterogeneous computing systems. In GPUs, a cache policy does not directly affect the performance due to presence of deep-multithreading. To estimate the effect of a cache behavior on GPU performance, they propose a core-sampling approach which is similar to set-sampling approach used in caches [60,61]. Since most GPU applications show symmetric behavior across the running cores, each core shows similar progress in terms of the number of retired instructions. Using this, core sampling applies a different policy (e.g. a cache replacement policy) to each core and periodically collects samples to see how the policies work. A large difference in performance of these cores indicates that GPU performance is affected by the cache policy. A negligible difference in performance shows that caching is not beneficial for this application. Using this, a decision about the best cache management policy can be made. Further, since the GPU has much larger number of threads than the CPU, GPU accesses the cache much more frequently than the CPU and the large number of accesses from GPU are likely to evict data brought in cache by the CPU threads. To address this issue, they introduce cache block lifetime normalization approach, which ensures that statistics collected for each application are normalized by the access rate of each application. Using this, along with a cache partitioning mechanism, cache can be intelligently partitioned between CPU and GPU, such that cache is allocated to GPU only if it benefits from the cache.

Mekkat et al. [42] propose a cache management policy for CPU-GPU heterogeneous computing systems with shared LLCs. Their technique leverages GPU's ability to tolerate memory access latency to throttle GPU LLC accesses to provide cache space to latency-sensitive CPU applications. Their technique works on the principle that the TLP available in an application is a good indicator of cache sensitivity of an application. Based on this, their technique allows GPU memory traffic to selectively bypass the shared LLC if GPU cores exhibit sufficient TLP to tolerate memory access latency or when GPU is not sensitive to LLC performance. The available TLP is measured at runtime using the number of wavefronts (or warps) that are ready to be scheduled at any given time. Higher number of wavefronts indicate higher TLP which suggests that GPU can tolerate higher memory access latency. Their technique uses core-sampling to apply two different bypassing thresholds to two different cores to find the impact of bypassing on GPU performance. Also, using cache set-sampling, the effect of GPU bypassing on CPU performance is estimated. Using these estimates, the rate of GPU bypassing is periodically adjusted.

Yang et al. [43] propose a technique for utilizing the idle CPU in a CPU-GPU

heterogeneous system to improve hit-rate of GPU threads in shared LLC (L3). In their technique, after the CPU launches a GPU program, it starts a pre-execution program to prefetch the off-chip memory data into the shared L3 cache for benefiting GPU threads. The pre-execution program is developed using a compiler algorithm and it extracts memory access instructions and the associated address computations from GPU kernels. Since CPU runs at higher frequency and leverages ILP (instruction-level parallelism) more aggressively, the pre-execution warms the shared L3 cache for GPU threads, which significantly reduces the memory access latency. Periodically, the timing of prefetches is adjusted to avoid cache pollution and increase the effectiveness of prefetching.

4.5. *Cache Management Policies For Improving Energy Efficiency*

Wang et al. [55] propose a technique for saving static (leakage) energy in both L1 and L2 caches in GPUs. They propose putting L1 cache (which is private to each core) in state-preserving[e] low-leakage mode when there are no threads which ready to be scheduled. Also, L2 cache is transitioned to low-leakage mode when there is no memory request. They also discuss the micro-architectural optimizations using which the latency of detecting cache inactivity and transitioning a cache to low-power and back to normal power can be completely hidden.

4.6. *Other Aspects*

Some researchers deduce the parameters of GPU caches through microbenchmarking and conducting experiments with specialized benchmarks [63,64,65,66]. This is very useful, since parameters for GPU caches are generally not publicly available. Moreover, it is also useful for developing GPU power/performance model and studying cache/memory behavior in isolation.

5. Conclusion

Multi-level hardware-managed caches are relatively recent addition to GPUs which also marks a paradigm shift in GPU architecture towards mainstream computing. Effective management of caches is vital to fully exploit their potential in boosting GPU performance and energy efficiency. In this paper, we presented a survey of system-level and architectural techniques for managing and leveraging GPU caches. We also presented a classification of the techniques based on their characteristics and optimization goals. We strongly believe that this paper will provide insights to the architects into working of GPU cache management techniques and also encourage them to propose novel techniques for GPUs of tomorrows.

[e]State-preserving or state-retentive leakage saving mechanism refers to use of a low-power state where the data stored in the block are not lost [62]. This is in contrast with state-destroying leakage control mechanism where the block data are lost in the low-power mode [62].

References

1. D. Patterson, "The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges," *NVIDIA Whitepaper*, 2009.
2. X. Cui, Y. Chen, C. Zhang, and H. Mei, "Auto-tuning dense matrix multiplication for GPGPU with cache," in *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*, 2010, pp. 237–242.
3. A. Schäfer and D. Fey, "High performance stencil code algorithms for gpgpus," *Procedia Computer Science*, vol. 4, pp. 2027–2036, 2011.
4. N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, 2006, pp. 6–6.
5. Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Using Fermi architecture knowledge to speed up CUDA and OpenCL programs," in *10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2012, pp. 617–624.
6. H.-K. Kuo, T.-K. Yen, B.-C. C. Lai, and J.-Y. Jou, "Cache Capacity Aware Thread Scheduling for Irregular Memory Access on Many-Core GPGPUs," in *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2013, pp. 338–343.
7. J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 11–22.
8. S. Ristov, M. Gusev, L. Djinevski, and S. Arsenovski, "Performance impact of reconfigurable L1 cache on GPU devices," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2013, pp. 507–510.
9. K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 133–137.
10. Z. Zheng and K. Mueller, "Cache-aware GPU memory scheduling scheme for CT backprojection," in *Nuclear Science Symposium Conference Record (NSS/MIC)*. IEEE, 2010, pp. 2248–2251.
11. G. J. Katz and J. T. Kider, Jr, "All-pairs Shortest-paths for Large Graphs on the GPU," in *23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH '08. Eurographics Association, 2008, pp. 47–55.
12. B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *International Symposium on Computer Architecture (ISCA)*, 2013.
13. C. J. Hughes, C. Kim, and Y.-K. Chen, "Performance and energy implications of many-core caches for throughput computing," *Micro, IEEE*, vol. 30, no. 6, pp. 25–35, 2010.
14. E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge, "Next-generation acceleration and code optimization for light transport in turbid media using GPUs," *Biomedical optics express*, vol. 1, no. 2, p. 658, 2010.
15. J. Nickolls and W. J. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.
16. Intel, http://download.intel.com/newsroom/archive/Intel-Itanium-processor-9500_ProductBrief.pdf.
17. A. Heinecke, M. Klemm, and H. Bungartz, "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture," *Computing in Science & Engineering*, vol. 14, no. 2, pp. 78–83, 2012.
18. J. Lee and H. Kim, "TAP: A TLP-aware cache management policy for a CPU-GPU

heterogeneous architecture," in *18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–12.
19. H. Choi, J. Ahn, and W. Sung, "Reducing off-chip memory traffic by selective cache management scheme in GPGPUs," in *5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, 2012, pp. 110–119.
20. I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for GPU architectures," in *HPCA*, 2013, pp. 578–590.
21. S. Mu, Y. Deng, Y. Chen, H. Li, J. Pan, W. Zhang, and Z. Wang, "Orchestrating cache management and memory scheduling for gpgpu applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2013.
22. T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.
23. H. Pirk, S. Manegold, and M. Kersten, "Accelerating foreign-key joins using asymmetric memory channels," in *VLDB-Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2011, pp. 585–597.
24. E. Blem, M. Sinclair, and K. Sankaralingam, "Challenge benchmarks that must be conquered to sustain the GPU revolution," *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture*, 2011.
25. W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *26th ACM international conference on Supercomputing*, 2012, pp. 15–24.
26. I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–12.
27. Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Understanding the impact of CUDA tuning techniques for Fermi," in *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2011, pp. 631–639.
28. T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.
29. O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13, 2013, pp. 157–166.
30. Y. Zhang, "Performance and Power Comparisons Between Fermi and Cypress GPUs," Louisiana State University, Tech. Rep., 2013, master's Thesis.
31. H. Choi, K. Hwang, J. Ahn, and W. Sung, "A simulation-based study for DRAM power reduction strategies in GPGPUs," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2012, pp. 1343–1346.
32. V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 451–460.
33. W. W. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 25–36.
34. S. Xiao, H. Lin, and W.-c. Feng, "Accelerating protein sequence search in a heterogeneous computing system," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)1*, 2011, pp. 1212–1222.
35. D. P. Playne and K. A. Hawick, "Comparison of GPU architectures for asynchronous communication with finite-differencing applications," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 1, pp. 73–83, 2012.

36. S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 78–88.
37. J. Wu and J. JaJa, "Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs," in *Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–12.
38. D. Cederman, B. Chatterjee, and P. Tsigas, "Understanding the performance of concurrent data structures on graphics processors," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Springer-Verlag, 2012, pp. 883–894.
39. L. H. Lourenço, D. Weingaertner, and E. Todt, "Efficient implementation of Canny Edge Detection Filter for ITK using CUDA," in *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*. IEEE, 2012, pp. 33–40.
40. S. Franey and M. Lipasti, "Accelerating atomic operations on GPGPUs," in *Seventh IEEE/ACM International Symposium on Networks on Chip (NoCS)*, 2013, pp. 1–8.
41. M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011, pp. 264–266.
42. V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing shared last-level cache in a heterogeneous multicore processor," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 225–234.
43. Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures," in *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.
44. J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.
45. A. Sankaranarayanan, E. Ardestani, J. Briz, and J. Renau, "An energy efficient GPGPU memory hierarchy with tiny incoherent caches," in *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2013, pp. 9–14.
46. M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 96–106.
47. A. Lashgar, A. Baniasadi, and A. Khonsari, "Inter-Warp Instruction Temporal Locality in Deep-Multithreaded GPUs," in *Architecture of Computing Systems–ARCS*. Springer Berlin Heidelberg, 2013, pp. 134–146.
48. I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm," *Concurrency and Computation: Practice and Experience*, 2012.
49. T.-K. Yen, H.-K. Kuo, and B.-C. Lai, "A distributed thread scheduler for dynamic multithreading on throughput processors," in *International Symposium on VLSI Design, Automation, and Test (VLSI-DAT)*, 2013, pp. 1–4.
50. A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *18th International conference on Architectural support for programming languages and operating systems (ASPLOS)*. ACM, 2013, pp. 395–406.
51. A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 332–343.

52. Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, 2009.
53. J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010, pp. 235–246.
54. A. Al Maashri, G. Sun, X. Dong, V. Narayanan, and Y. Xie, "3D GPU architecture using cache stacking: Performance, cost, power and thermal analysis," in *IEEE International Conference on Computer Design (ICCD)*, 2009, pp. 254–259.
55. Y. Wang, S. Roy, and N. Ranganathan, "Run-time power-gating in caches of GPUs for leakage energy savings," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 300 –303.
56. J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *IEEE International Conference on Computer Design (ICCD)*, 2009, pp. 282–288.
57. M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.
58. S. Mittal, *Architectural Techniques For Managing Non-volatile Caches*. Germany: Lambert Academic Publishing (LAP), 2013.
59. M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *International Symposium on Microarchitecture (MICRO)*, 2013.
60. S. Mittal, "Dynamic cache reconfiguration based techniques for improving cache energy efficiency," Ph.D. dissertation, Iowa State University, 2013.
61. T. Puzak, "Cache memory design," Ph.D. dissertation, University of Massachusetts, 1985.
62. S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustainable Computing: Informatics and Systems*, 2013.
63. H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010, pp. 235–246.
64. V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *ACM/IEEE conference on Supercomputing*, 2008, p. 31.
65. J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 487–498.
66. J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 11–22.