Techniques for Caches in GPUs

Guenther Schindler,
University of Heidelberg, ZITI,
G.Schindler@stud.uni-heidelberg.de

Abstract—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. In this paper, we analyze three recent proposals for managing and leveraging GPU caches. The first proposal presents a mechanism for implementing low-cost coherence and speculative acquisition of atomic data on the GPU. The second proposal introduces two cache management schemes: write-buffering and read- bypassing. The write buffering technique tries to utilize the shared cache for inter-block communication, and thereby reduces the DRAM accesses as much as the capacity of the cache. The read-bypassing scheme prevents the shared cache from being polluted by streamed data that are consumed only within a thread-block. The last proposal introduces a cache management policy of CPU-GPU heterogeneous computing systems.

Index Terms—GPU, GPGPU, cache memory, energy efficiency, cache replacement decision, off-chip memory traffic, Atomic Operations, cache bypassing, heterogeneous architecture.

1 Introduction

Graphics Processing Units (GPUs), the coprocessor originally designed predominantly for graphic rendering, nowadays has been proven unexpectedly successful in the domain of general-purpose applications (GPGPU). The demands of new application domains have motivated novel changes in GPU 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 caches. While CPU cache management has been studied over years, GPU cache management is a relatively new research field.

In this paper, we present and analyze three recent proposals of techniques for managing and leveraging GPU caches. The first one is motivated by parallel workloads that utilize atomic operations to update globally shared data, particularly those that utilize these updates to perform coarse, global synchronization. Modern GPUs do not efficiently support synchronization outside of very local work units. There is no direct support for synchronizing threads across computational blocks. GPGPU application have had to rely exclusively on atomic operations to global data. While this is a common technique utilized in general purpose CPUs, the lack of L1 cache coherence in GPUs make these operations significantly slower.

The second technique is motivated by the fact that GPGPUs demand a very high external memory bandwidth. Raising the off-chip memory bandwidth by the increased clock frequency or pin count is considered very difficult because of the energy wall problem and the package limitation. As a result, the off-chip memory bandwidth becomes a critical performance limiting factor and a significant source of energy and power consumption as well. One solution to this growing problem is to reduce the number of off-chip memory accesses by using on-chip memory or cache. Selective cache management schemes control the placement of data in the shared L2 cache to reduce the memory traffic as much as possible when executing GPGPU programs.

The last proposal introduces a cache management policy of CPU-GPU heterogeneous computing systems. Based on the fact that on-chip heterogeneous architectures have become a new trend. In particular, combining CPUs and GPUs is one of the major trends. In these architectures, various resources are shared between CPUs and GPUs, such as the last-level cache, on-chip interconnection, memory controllers, and off-chip DRAM memory. The last-level cache (LLC) is one of the most important shared resources in chip multi-processors. Managing the LLC significantly affects the performance of each application as well as the overall system throughput.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the GPU architecture and a comparision between GPU and CPU caches. Section 3 introduces mechanisms to improve atomic operations in GPUs. Section 4 presents a scheme to avoid off-chip accesses for inter-block communication. Section 5 shows two examples for cache bypassing in CPU-GPU heterogeneous systems. Finally section 6 presents the conclusion.

2 Background

In this section, we first briefly introduce the most important features of the GPU architecture as well as the execution model. We then describe the differences between GPU caches and CPU caches. Although we focus on NVIDIA GPUs and use CUDA terminology in the paper, the concepts also apply to AMD GPUs.

2.1 GPU Architecture

Based on SIMD, the execution model of GPUs is named single-instruction-multiple-threads or SIMT [7]. A kernel,

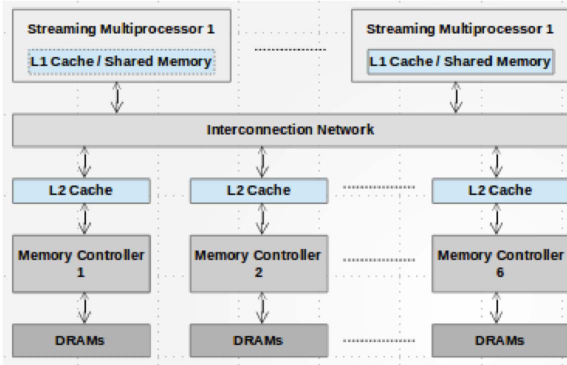

Fig. 1. Memory hierarchy of Fermi architecture.

which is a function that runs on the GPU, includes thousands of threads that are grouped into multiple thread blocks (cooperative thread arrays (CTAs)). This CTAs will be mapped to several streaming multiprocessors (SMs) when a kernel launch occurs. Threads inside a CTA are further organized in groups of 16 or 32, called warps. In an SM, a warp is the basic unit in terms of scheduling, executing and accessing memory. If a warp is obstructed by a long latency operation the warp scheduler will switch to another ready warp instantly with no cost [7].

The GPU memory architecture contains registers, L1 cache, read-only data cache, interconnection network, L2 cache and global memory. Registers are private to threads. The L1 and read-only caches are shared by all CTAs in an SM. The L1 cache is not coherent. Furthermore, the L1 cache shares the same on-chip storage with the shared memory (or Scratch Pad) of an SM. Their relative sizes are reconfigurable. Scratch Pad is a software controlled cache which is explicitly controlled by the programmer. SMs are connected to a unified L2 cache by an interconnection network. The L2 cache is generally partitioned into several banks (Figure 1), each of them being a buffer for GDDR memory channels. This partitioning makes the L2 cache implicitly coherent.

One of the most important facts about GPU memory is coalescing. Simultaneous memory request from threads in the same warp are usually combined as a group request for a cache-line sized chunk before accessing L1 cache. Non-coalesce memory accesses will end up in a significant loss of bandwidth.

The L1 cache line is 128B and caches global memory read accesses. The L2 cache is much larger with a smaller cache line size of 32B and serves all types of memory accesses. [7]

2.2 GPU Caches vs. CPU Caches

GPU chips spend more die-space on ALUs and less on caches whereas CPU chips have huge caches and just a few ALUs. For example, Intel's Itanium 9560 CPU uses 32MB last level cache (LLC) [3]. 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 [4]. Furthermore, the per-thread cache share for GPUs is much smaller than for CPUs, which indicates that the useful data fetched by one thread is very likely to be evicted by other threads before actual re-usage. For example, Intel's Haswell has 16KB L1 cache per thread. Nvidia's Fermi has only 32B and and Kepler 24B L1 cache per thread [5]. Thus, caches in GPUs can take advantage of spatial locality but not of temporal locality. Instead of seeking performance via out-of-order processing and large caches, GPUs make use of low-overhead thread scheduling and hide memory latencies via multi-threading. CPUs are low latency low throughput processors whereas GPUs are high latency high throughput processors.

Unlike CPU architectures from Intel and AMD, the L1 caches in CUDA capable cards are not coherent. This means that if two different SMs are reading and writing to the same memory location, there is no guarantee that one SM will immediately see the changes from the other SM.

3 Atomic Operations

As new applications are explored and highly parallel algorithms are adapted to GPUs, the demand for new features is growing. A large portion of applications are still not suited for GPUs due to their high utilization of atomic operations, particularly as global synchronization mechanism. There is no direct support for synchronizing threads across CTAs. Unlike the sophisticated synchronization mechanisms in CPUs or cluster-systems, GPU applications must rely on slow atomic operations on shared data. CPUs, for example, take advantage of coherent L1 caches to efficiently implement atomic operations. These traditional implementations are not possible on GPUs due to the lack of L1 cache coherency. Implementing coherency on GPUs seems to be impossible because of the high design and verification costs, coupled with the fact that the primary use of GPUs - rendering of graphics - does not benefit from it. Therefore, it is inevitable to implement inexpensive mechanisms which fulfill this demands.

In this section, we first analyze the current state-of-the-art regarding atomic operations on the GPU. We then present mechanisms to provide high speed atomic operations to applications on the GPU, published by Franey et al. [12].

3.1 Present Atomic Operations

In order to understand solutions for atomic operations, one must have an understanding of the current implementation on GPUs. Unfortunately implementation of atomic operations on GPUs have not been publicly described but Franey et al. assume that atomic instructions are executed like non-atomic instructions in the shader core. According to that, an atomic operation is generated in the shader core and traverses the interconnect to the appropriate L2 bank. Once at the L2 bank, the operation is ordered, data is acquired, and the operation is performed. The new data is written back and a response is sent back to the core containing the previous value of the data. [12]

In order to avoid the latency of traversing the interconnect, the atomic operations must be performed locally with local data.

3.2 AtomDir

The first mechanism is a simple adaption of Atomic Coherence [13] that allows the GPU to implement L1 cache

coherence on atomic data. Atomic Coherence prevents race condition by mimicking a traditional blocking bus. This is originally realized by leveraging a nanophotonic ring that is unique to the system it is evaluated on [13]. The ring topology must connect all nodes that would need to acquire mutexes. Implemented by a rotating token this ring need not to be a physical set of wires. This token can be realized by a modulo operation on the cycle count. A node knows that is the holder of the token based on how many time-steps have occurred. If a node acquires a mutex it simply has to wait for the token, can then (assuming the mutex is available) acquire the mutex and mark the it unavailable for subsequent requesters. Releasing the mutex follows the same scheme.

Regarding this implementation it is necessary to add a structure to each node to track the state of the mutexes and buffers to hold requests until the token arrives. Franey et al. introduced a mechanisms called ”busy wire” to indicate to nodes when an update is in flight. The busy wire is a point-to-point wire using the underlying interconnect and is conceptually running along with the data links. When the busy wire signals an update all mutex acquisitions are stalled. If it signals no update acquisitions are allowed to occur. Each node is responsible for keeping the busy wire asserted until that node indicates it no longer needs to be asserted. The state of the busy wire then propagates through the system at a rate at least equal to the rotating token to ensure no mutexes are acquired until the update completes. It is also necessary to add more than one busy wire in order to reduce false conflict problems. The amount of additional wires is a design decision based on how many mutexes should be available in the system.

However, this mechanism called AtomNaive performed not much better than the current state-of-the-art approach and much worse for certain applications. This is due to the long latency to acquire the token and the additional latency for updates.

In order to improve this performance Franey et al. replaced the token by adapting techniques used in directory-based cache coherence. By replacing the token by a ”owner” directory and remove the updates by unique home nodes the overall latency should be reduced. When a node acquires a mutex, it simply requests the mutex from the owner instead of waiting for the token. If the mutex is available, the owner then responds it back to the requester. This mechanism, called AtomDir, replaces the waiting for the token with a round-trip communication with the owner.

3.3 Hybrid Topology

Further, Franey et al. extend the previous described mechanisms to improve latency motivated by the fact that latency is optimal when a node is able to satisfy its own mutex requests in the same cycle the requests are generated (self-satisfied request). In the AtomDir case, this occurs only when the requester happens to also be the owner of the mutex. In AtomNaive, this occurs whenever the talking stick is present at the requester on the same cycle of the request. Therefore, we attempt to increase the number of self-satisfied mutex requests - that is maximized in AtomNaive - without imposing the severe latency overhead of the token. This is done by implementing a system that is hybrid of the single owner directory of AtomDir and the fully-distributed directory in AtomNaive. Here, the mutex state is distributed across different logical rings, where each member of a ring maintains and responds to requests for the same mutexes and nodes in different rings maintain different mutexes.

This hybrid implementation can reduce the latency because it replaces the Δx plus Δy latency to transmit a request to an owner with the token latency in one dimension (Δx, nominally). This is beneficial because average token latency is half Δx since it is alway a one-way trip while Δx is a round-trip communication.

3.4 Speculative Fetches

In the state-of-the-art implementation of atomic operations they simply traverse the interconnect to the L2 and are immediately satisfied, while in the mechanisms, presented by Franey et al., they incur the additional mutex acquisition penalty. Therefor, they introduced a final optimization to support speculative fetches which allows to fetch memory in parallel with a mutex request. To ensure correctness of the fetched data they use a method called epochs in order to differentiate mature in immature events in the system. An epoch consists of a fixed number of cycles. At the boundary of each epoch, all responders indicate that their mutex releases (i.e., available mutexes in the mutex status table) are mature and all requesters indicate that their outstanding mutex requests are stale. Therefore when a responder sends a mutex to a requester, it indicates whether the last release was mature or not. When the requester receives the mutex, if it is mature, it checks whether or not its request is stale. If both the release is mature and the request is not stale, the requesting node knows that no update could have occurred to the data associated with a mutex between its speculative fetch and mutex acquisition (i.e., both conditions of the release being done in a previous epoch and the request being made in the current epoch are satisfied).

3.5 Evaluation

In order to evaluate the contribution of the different mechanisms (AtomDir, Topology and Speculative Fetch), Franey et al. simulated the system, building features on top of one another.

As in Figure 8 can be seen, AtomDir shows the benefit of being able to cache atomic data, Topology shows the benefit of distributing ownership, and SpecFetch shows the advantage of issuing speculative memory fetches along with mutex acquisition.

4 Communication Through Caches

Since Fermi the L2 cache is used to provide a buffer for inter-block communication. Some GPU applications suffer from the lack of an efficient inter-block synchronization mechanism. A widely used solution to this problem is to exit the current kernel and re-launch the successive kernel after a global synchronization by the host.

In this section we analyze the current communication scheme, identify its problems and limitations, and present a

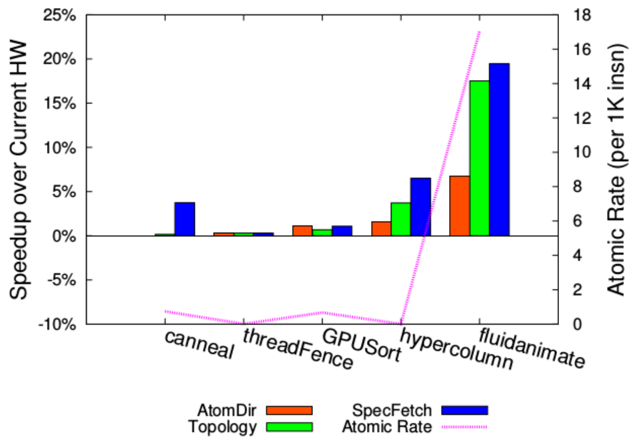

Fig. 2. Contributions to Performance. [12]

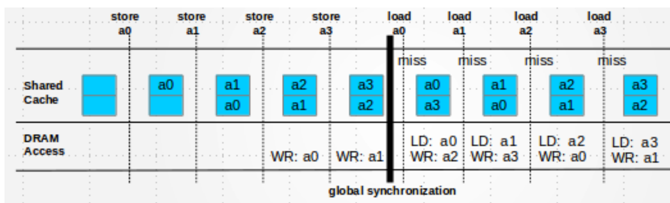

Fig. 3. The worst case usage of the LRU replacement policy during inter-block communication.

mechanism, introduced by Choi et al. [6], which improves inter-block communication.

A inter-block communication can be performed by load and store instructions without demanding off-chip memory transactions when the cache size is large enough to hold all of the data involved. However, when the cache size is smaller than the working set, the L2 cache with a simple management scheme can not reduce the number of potentially unnecessary off-chip memory accesses during the inter-block communication.

Figure 3 shows an extreme case of the LRU replacement policy, where the shared L2 cache does not help reducing the off-chip memory traffic for inter-block communication.

In this example, four addresses, denoted as a0-a3, are accessed for store and load operations in two consecutive kernels, respectively. It is assumed that the shared cache has only single set of two ways for simplification, where the lower box means the LRU position. As shown in figure 3, the cache lines allocated by write operations are evicted by other write operations of the same kernel before the global synchronization. Some of the cache lines allocated for write operations may remain at the kernel completion, but they will be evicted completely by read operations of the successive kernel. As a result, the amount of off-chip memory accesses is the same, whether there is L2 cache or not.

4.1 Write-buffering for inter-block communication

With the LRU replacement policy, the L2 cache works as a FIFO (First-In First-Out). Choi et al. [6] prevent this by modifying the cache management scheme. For that purpose,

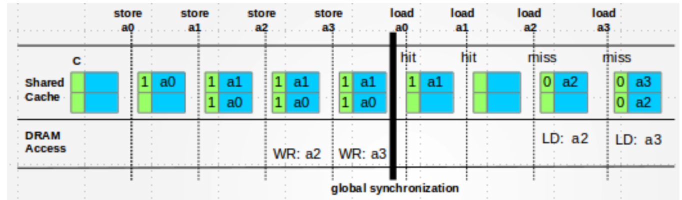

Fig. 4. An example of write-buffering.

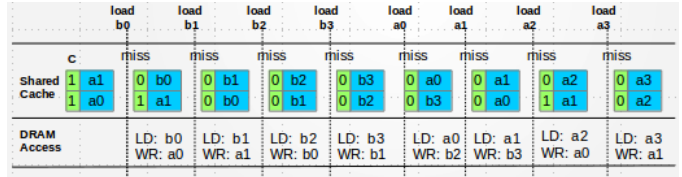

Fig. 5. Write-buffering without read-bypassing.

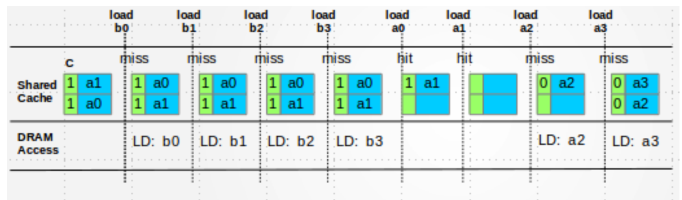

Fig. 6. Write-buffering with read-bypassing.

a 1-bit status flag, denoted as C, is added to every cache line. On a write miss, if the write operation is directed to the write-buffering, a cache line is allocated and C is set. During the allocation, if the C flag of a cache line is set, the line is not selected for replacement. If the flag for every cache line is set, then the write reference bypasses the L2 cache and is forwarded to off-chip memory without allocation. The reason is that the L2 cache otherwise works as a FIFO. When a read hit occurs for the line with which the C flag is set, the line is invalidated immediately to make it available for replacement without any off-chip memory access.

As can be seen in figure 4, the write-buffering prevents the cache lines for a0 and a1 from being replaced by a2 or a3. Since they are retained until being read after the global synchronization, the number of DRAM accesses is reduced.

4.2 Read-bypassing for private data

Figure 5 shows a negative example of write-buffering. If load operations are made for b0-b3, they may evict the cache lines for a0 and a1 due to conflict or capacity misses. Thus there is no benefit of applying the write-buffering and the number of off-chip memory access is the same with that of the pure LRU replacement policy.

Choi et al. [6] proposed another cache management scheme, named read-bypassing. If the read-bypassing is applied to private data load operations for b0-b3, the shared cache simply bypasses them to upper-level memory and does not allocate the cache lines for them. Figure 6 shows that the cache lines allocated for a0-a1 with the write-buffering can be retained until they are read.

The proposed scheme is software-controlled where load and store instructions are marked with their respective

scheme. Global load and store instructions can be annotated with cache operators which control the L2 cache behavior [7]. Currently, the cache operators are applied to all of the load/store instructions in a program by the PTX assembler options. Two additional cache operators, named .cc and .cp, are defined for write-buffering and read-bypassing. The load instruction directed with .cp bypasses L2 cache. For a store instruction directed with either .cc, a cache line is allocated on a cache miss an the C bit is set for write buffering.

4.2.1 Performance Evaluation

Choi et al. [6] used FFT, HotSpot and SRAD as workloads, which all show inter-block communication, as workloads. The performance of the proposed technique is showed in figure 7 using the example of FFT.

As a result of the proposed technique, the amount of write traffic to the off-chip memory is reduced. The number of off-chip read accesses is also reduced since the cache lines allocated for the write-buffering serve the read accesses of the successive kernel.

5 Cache Bypassing

Combining CPUs and GPUs on the same chip has become a popular architectural trend, as can be seen from Intels recent Sandy Bridge, AMDs Fusion, and NVIDIAs Denver. In these architectures, various resources are shared between CPUs and GPUs, such as the last-level cache (LLC), on-chip interconnection, memory controllers, and off-chip DRAM memory. However, such heterogeneous architectures put more pressure on shared resource management. In particular, managing the LLC is very critical to performance.

Under the LRU approximations, widely used in modern caches, applications that have high cache demand acquire more cache space. The easiest example of such an application is a streaming application. Even though a streaming application does not require caching due to the lack of data reuse, data from such an application will occupy the entire cache space under LRU when it is running with a non-streaming application. Thus, the performance of a non-streaming application running with a streaming application will be significantly degraded.

In order to improve the overall performance of caches, researchers have proposed a cache mechanisms to identifies the dominant pattern within an application and avoids caching for non-temporal data. This can be done by inserting incoming cache blocks into positions other than the most recently used (MRU) position to enforce a shorter life time in the cache.

5.1 TLP-Aware Cache Management Policy

This section proposes a thread-level parallelism-aware cache management policy (TAP), introduced by Kim et al. [8], that consists of two components: core sampling and cache block lifetime normalization. They also propose two new TAP mechanisms: TAP-UCP and TAP-RRIP.

Based on the need of a way to identify the cache-to-performance effect for GPGPU application they propose a sampling mechanism that applies a different policy to each core, called core sampling. Core sampling applies a different policy to each core and periodically collects samples to see how the policies work. For example, to identify the effect of cache on performance, core sampling enforces one core to use the LRU insertion policy and another core (Core-POL2) to use the MRU insertion policy. Once a period is over, the core sampling controller (CSC) collects the performance metrics, such as the number of retired instructions, from each core and compares them. If the CSC observes significant performance differences between Core-1 and Core-2, we can conclude that the performance of this application has been affected by the cache behavior.

GPGPU applications typically access caches much more frequently than CPU applications. Even though memory-intensive CPU applications also exist, the cache access rate cannot be as high as that of GPGPU applications due to a much smaller number of threads in a CPU core. Also, since GPGPU applications can maintain high throughput because of the abundant TLP in them, there will be continuous cache accesses. However, memory-intensive CPU applications cannot maintain such high throughput due to the limited TLP in them, which leads to less frequent cache accesses. As a result, there is often an order of difference in cache access frequencies between CPU and GPGPU applications. Hence, when CPU and GPGPU applications run together, we have to take into account this difference in the degree of access rates.

To solve this issue, Kim et al. [8] introduced cache block lifetime normalization. First, they detect access rate differences by collecting the number of cache accesses from each application. Periodically, they calculate the access ratio between applications. The TAP policies utilize the value of this ratio to enforce similar cache residential time to CPU and GPGPU applications.

5.1.1 TAP-UCP

TAP-UCP is based on UCP [9], a dynamic cache partitioning mechanism for only CPU workloads. Base on the observation that not all threads/applications benefit equally from caching (simple LRU replacement is not good for system throughput) the idea of UCP is to allocate more cache space to applications that obtain the most benefit from more space. UCP periodically calculates an optimal partition to adapt a run-time behavior of the system.

To apply TAP in UCP, we need two modifications in the UCPs partitioning algorithm. The first modification is that when the CSC identifies that caching is not effective, only CPU applications are considered for cache allocation. The second modification is that partitioning is performed based on the cache access ratio which is periodically set by cache block lifetime normalization.

5.1.2 TAP-RRIP

The Re-Reference Interval Prediction (RRIP) mechanism [10], which is the base of TAP-RRIP, dynamically adapts between two competing cache insertion policies, Static-RRIP (SRRIP) and Bimodal-RRIP (BRRIP) to filter out trashing patterns. A saturating counter, called a Policy Selector (PSEL), keeps track of which policy incurs fewer cache misses and decides the winning policy.

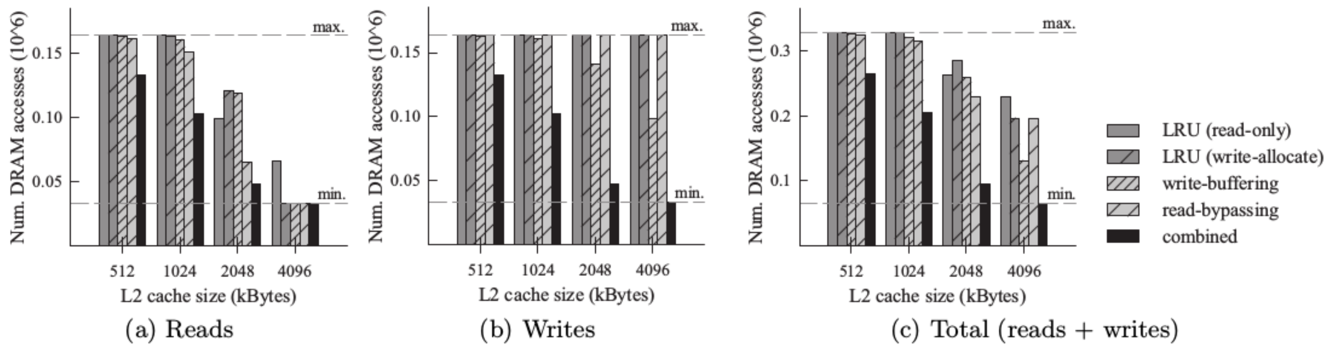

Fig. 7. Effect on the off-chip memory traffic reduction in FFT.

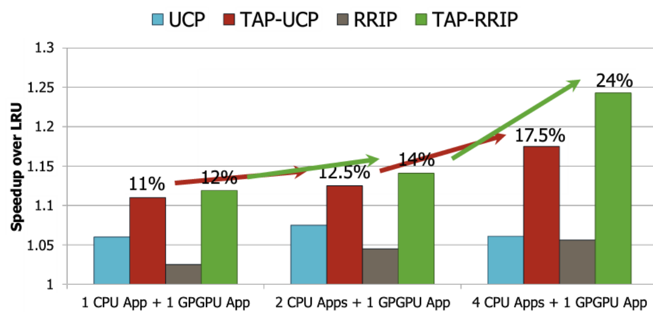

Fig. 8. Performance of TAP. [11]

When the CSC identifies that caching is not effective TAP-RRIP enforce the BRRIP policy for the GPGPU application since BRRIP generally enforces a shorter cache lifetime than SRRIP for each block. Further, if the value of the cache access ratio (GPU/CPU) is greater than 1, the policy for the GPGPU application will be also set to BRRIP. Otherwise, the winning policy by PSEL will be applied.

5.1.3 *Performance Evaluation*

Kim et al. [8] evaluate the TAP mechanisms on 152 heterogeneous workloads and showed that they improve the performance by 5% and 10% compared to UCP and RRIP and 11% and 12% to LRU. Also, they showed that TAP mechanisms show higher benefits with more CPU applications.

6 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 very important to fully exploit their potential in boosting GPU performance and energy efficiency. In this paper, we presented three recent proposals by different researchers.

We have presented a low-latency mechanism, proposed by Franey et al. [12], for acquiring and releasing mutexes in a system of multiple nodes, applied to improve the performance of atomic operations on a GPU. This mechanisms achieve modest improvements for GPGPU workloads on Fermi. Still, it is very doubtful that it improves performance on later GPU architectures since Kepler already reduced the L2 latency significantly and therefore the performance of atomic operations. Besides, Maxwell only has a L1 read-only cache so it is likely that this mechanism can't be applied there.

We have analyzed the publication by Choi et al. [6] which shows that the off-chip memory accesses can be substantially reduced by the proposed techniques, namely write-buffering and read-bypassing. Larger L2 caches are obviously a trend in GPUs. For example Fermi implements 768 KB whereas Maxwell implements 2048 KB. The performance evaluation shows, that the proposed mechanism improves more with larger L2 caches. Also, latency of L2 caches in GPUs decreases which should improve performance further. A negative aspect is the high implementation cost for programmers which have to add the instructions manually.

In order to identify the characteristics of a GPGPU application in heterogeneous processors, we finally analyzed a proposed mechanism, published by Kim et al. [8], called core sampling, which is a simple yet effective technique to profile a GPGPU application at run-time. By applying core sampling to UCP and RRIP and considering the different degree of access rates, the researchers proposed the TAP-UCP and TAP-RRIP mechanisms. The TAP mechanism shows good performance improvements over LRU and is not restricted to specific architectures. This comes with the cost of high overhead for control logic and storage. However, none of the previous mechanisms consider GPGPU-specific characteristics in heterogeneous workloads.

We believe that this paper will provide the readers insights into GPU cache management techniques and motivate them to propose even better techniques for leveraging the full potential of caches in the GPUs of tomorrow.

References

[1] Future Technologies Group, Oak Ridge National Laboratory (ORNL), ”A Survey of Techniques for Managing and Leveraging Caches in GPUs”, Journal of Circuits, Systems, and Computers, 2014.
[2] Nvidia. CUDA Programming Guide. 2015.
[3] Intel, http://download.intel.com/newsroom/archive/Intel-Itanium-processor-9500 ProductBrief.pdf.
[4] 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. 7883, 2012.

[5] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal, ”Adaptive and Transparent Cache Bypassing for GPUs”, C 15, November 15-20, 2015, Austin, TX, USA.

[6] 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. 110119.

[7] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 2.0, 2010.

[8] 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. 112.

[9] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In MICRO-39, pages 423432, 2006.

[10] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In ISCA-32, pages 6071, 2010.

[11] J. Lee and H. Kim, ”TLP Aware Cache Management Policy”, Talk HPCA-18.

[12] S. Franey and M. Lipasti, Accelerating atomic operations on GPGPUs, in Seventh IEEE/ACM International Symposium on Networks on Chip (NoCS), 2013, pp. 18.

[13] D. Vantrease, M. Lipasti, and N. Binkert, Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols, in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, 2011, p. 132143.