Improving GPU Cache Hierarchy Performance with a Fetch and Replacement Cache

Francisco Candel[1(✉)], Salvador Petit[1], Alejandro Valero[2], and Julio Sahuquillo[1]

[1] Department of Computer Engineering, Universitat Politècnica de València, 46022 Valencia, Spain
fracanma@inf.upv.es, {spetit,jsahuqui}@disca.upv.es

[2] Departamento de Informática e Ingeniería de Sistemas, Instituto Universitario de Investigación en Ingeniería de Aragón Universidad de Zaragoza, 50018 Zaragoza, Spain
alvabre@unizar.es

Abstract. In the last few years, GPGPU computing has become one of the most popular computing paradigms in high-performance computers due to its excellent performance to power ratio. The memory requirements of GPGPU applications widely differ from the requirements of CPU counterparts. The amount of memory accesses is several orders of magnitude higher in GPU applications than in CPU applications, and they present disparate access patterns. Because of this fact, large and highly associative Last-Level Caches (LLCs) bring much lower performance gains in GPUs than in CPUs.

This paper presents a novel approach to manage LLC misses that efficiently improves LLC hit ratio, memory-level parallelism, and miss latencies in GPU systems. The proposed approach leverages a small additional Fetch and Replacement Cache (FRC) that stores control and coherence information of incoming blocks until they are fetched from main memory. Then, fetched blocks are swapped with victim blocks to be replaced in the LLC. After that, the eviction of victim blocks is performed from the FRC. This management approach improves performance due to three main reasons: (i) the lifetime of blocks being replaced is increased, (ii) the main memory path is unclogged on long bursts of LLC misses, and (iii) the average L2 miss delaying latency is reduced. Experimental results show that our proposal increases the performance (OPC) over 25% in most of the studied applications, reaching improvements up to 150% in some applications.

1 Introduction

In recent years, GPU (Graphics Processing Unit) architectures have acquired a great relevance in the field of high-performance computing. The main reason has been that GPUs are able to accelerate the execution of massively parallel

© Springer International Publishing AG, part of Springer Nature 2018
M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 235–248, 2018.
https://doi.org/10.1007/978-3-319-96983-1_17

applications, since they provide a much higher level of parallelism than CPU architectures. In addition, GPUs are energetically more efficient [1,2] for a given performance, than its CPU counterparts. Because of these reasons, many supercomputers in the top 500 list [3] rely on GPUs. For instance, the Piz Daint supercomputer, ranked in third place of the list in November 2017, was built with Nvidia Tesla P100 GPU devices.

GPU architectures are optimized to run applications composed of thousands of logical threads. In order to support the execution of such a high number of threads, the GPU core must be coupled with a memory subsystem able to support a high Memory-Level Parallelism (MLP). GPU memory subsystems are therefore designed to sustain a high memory bandwidth. Because of the poor data temporal locality of GPGPU applications or kernels, on a *very long* burst of L2 accesses many requests can miss, which cause subsequent main memory accesses.

In this scenario, the memory subsystem of GPUs poorly performs. In this paper, we look into the reasons explaining this behavior, and we find that one of the main sources of performance losses of the memory subsystem is the management of L2 cache misses. We find that conventional caches designed to address memory patterns of CPU applications do not properly meet the requirements of GPGPU applications, but they seriously penalize their performance since they can significantly slow down the management of L2 cache requests on long bursts of requests. The previous rationale means that improving the L2 cache management is a key design concern that should be tackled to improve the system performance. This paper proposes a novel L2 cache design aimed at boosting the memory level parallelism by adding a Fetch and Replacement Cache (FRC) that provides additional cache lines that help unclog the memory subsystem. The FRC approach uses these extra resources to prioritize the fetch of incoming L2 cache requests and to delay the eviction of the blocks to be replaced. The proposal has been evaluated considering an AMD GPU based architecture, although the results would also apply in almost all current GPU architectures as they implement a similar memory hierarchy.

The proposal has been modeled in the Multi2Sim simulation framework [4], a state-of-the-art GPU simulator widely used in both the academia and the industry. Experimental results show that FRC improves the Operations Per cycle (OPC) more than 25% in most applications by drastically reducing the Misses Per Kilo-Operation (MPKO) and L2 miss latency.

The remainder of this work is organized as follows. Section 2 describes the architecture of the AMD *Southern Islands* family of GPUs. Section 3 motivates this work by presenting the problems that FRC tackles in current GPU memory subsystems. In Sect. 4, the proposed approach is described in detail. Section 5 presents the experimental results. Section 6 summarizes related studies about GPU memory subsystems. Finally, in Sect. 7 some concluding remarks are drawn.

2 Background

This section provides some background about the architecture of modern GPUs. Since this paper focuses on the AMD *Southern Islands* [5] family of GPUs, AMD terminology is used throughout this work.

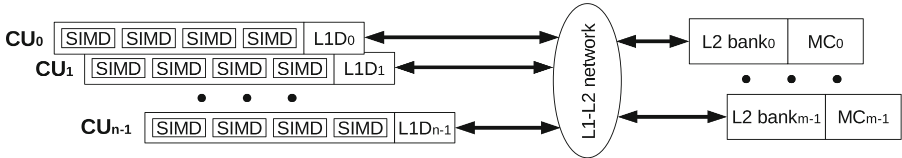

Fig. 1. Diagram of an AMD Southern Islands GPU.

Figure 1 depicts a block diagram of an AMD Southern Islands GPU. This GPU includes up to 32 *Compute Units* (CUs), each one implementing the *Graphics Core Next* (GCN) [6] microarchitecture. Internally, a GCN CU consists of 4 *Single Instruction Multiple Data* (SIMD) arithmetic logic units.

GPU applications or *kernels* are composed of a massive number of threads or *work-items*. These threads are organized in 64-thread bundles, named *wavefronts*, which are allocated to SIMD units. During most of the execution time of a kernel, the GPU ensures that each SIMD unit is assigned tens of wavefronts. In this way, SIMD units can switch among wavefronts in a fine-grain basis, which helps hide memory latencies.

A SIMD unit executes instructions from threads of a wavefront in a lockstep manner. That is, at a given point of the execution time a SIMD unit is performing the same arithmetic instruction in the 64 threads of the same wavefront. Memory reference instructions are also executed following the SIMD paradigm; that is, a wavefront can generate up to 64 memory requests at the same time. To reduce the overall amount of memory requests, those referencing the same 64-byte cache block are *coalesced* into a single memory request, which is issued to the memory subsystem.

As in a conventional processor, the memory subsystem is organized hierarchically. After being coalesced, memory requests access the L1 data cache of the corresponding CU. Those requests that miss the L1 cache are forwarded to a multi-banked L2 cache, acting as Last-Level Cache (LLC). L2 banks contain interleaved block addresses at a granularity of 256 bytes, and each bank is connected to a dual-channel memory controller that manages the corresponding off-chip GDDR5 main memory. This design reduces the number of channel conflicts and increases the memory bandwidth utilization.

3 Motivation

The coalesce mechanism reduces the number of requests to the memory subsystem. However, GPGPU applications generate enormous amounts of memory

traffic; for instance, a typical GPU can issue thousands of memory requests in a given cycle. These amounts yield conventional cache organizations to significant performance losses. The main reason is that the massive number of threads is executing in parallel causes sudden bursts of memory transactions, which involve a high number of cache replacements. As a consequence, in a relatively short interval of time, a given cache line can suffer a long number (e.g. in the order of tens) of consecutive block replacements, each one involving different actions such as coherence invalidations or accesses to lower levels of the memory hierarchy. Since these actions are serialized at the cache line, the management of cache replacements becomes a major performance bottleneck, which can heavily reduce the MLP and the L2 hit ratio.

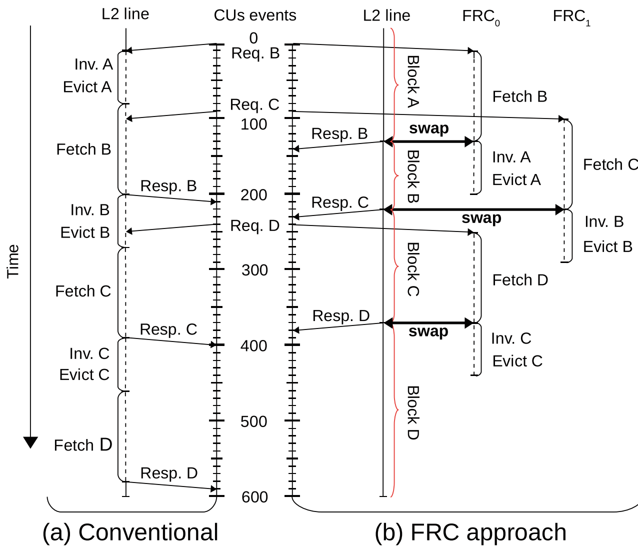

Fig. 2. Sequence of events involved in three consecutive replacements targeting the same L2 cache line for both the conventional and the proposed approaches.

To help understand the problem, Fig. 2 depicts a time diagram with the events involved in three consecutive replacements all targeting the same L2 victim line. The three requests causing these replacements have been labeled as *Req. B*, *C*, and *D*, and have been generated at cycles 0, 90, and 240, respectively, after the requests miss the L1 cache and are forwarded to the L2 cache.

As can be seen in Fig. 2a, which shows the behavior of a conventional replacement approach, *Req. B* triggers the replacement of the currently stored block (block *A*). From this moment, the victim line is in a transient state (represented by dashed lines), preventing other requests from accessing the line. To manage the replacement, depending on the state of *A*, an invalidation to the L1 cache and an L2 cache eviction must be performed. Once the victim line is freed, the

requested incoming block (B), must be fetched from main memory and allocated to this line.

While block B is being fetched, *Req. C* arrives to L2, which triggers another replacement in the same victim line. However, because of the line is in a transient state, *Req. C* must be enqueued. Thus, *Req. C* cannot be attended until cycle 210, delaying its completion until cycle 400. This serialization also affects *Req. D* at cycle 240.

Moreover, the hit ratio is also reduced, since (i) the invalidation and eviction of the contents of a victim line are performed before fetching the requested block and (ii) the fetch operation is the longest one involved in a replacement due to the high main memory latencies. As an example, even if a complex protocol allows reading the contents of a cache line while it is in a transient state, a load requesting block A would only hit between cycles 0 and 90, and would miss afterwards.

Although theoretically possible, it is very rare that this situation occurs in a conventional CPU processor since there is likely a non-transient line in the same cache set that can be selected as a victim, which avoids the serialization of replacements. In contrast, in GPUs, it is often the case that a burst of misses triggers replacements in all the lines of the same cache set. Therefore, further misses targeting the same set cannot be served from memory, which impacts on the exploited memory parallelism.

A naive solution to this problem is blindly increasing cache associativity so that a set has more available lines. However, this approach incurs in high latencies and energy penalties since associative tag lookups do not scale well with the number of ways. Moreover, although such a solution may alleviate the problem, larger sets can also be blocked provided that bursts of misses affecting the same cache set are large enough.

4 FRC Approach

The proposed approach is aimed at increasing MLP and LLC hit ratio. With this aim, we introduce a Fetch and Replacement Cache (FRC) to each L2 cache bank. The FRC provides additional cache lines that allow (i) start fetching from memory as soon as an L2 miss rises, increasing MLP, and (ii) performing invalidation and eviction actions *after* fetching the requested block, which increases the lifetime of victim blocks and the overall hit ratio.

Figure 2b shows how the FRC can help improve the management of consecutive replacements affecting the same line. By cycle 10, when *Req. B* misses in L2, instead of immediately invalidating the victim line, a free FRC entry (FRC_0) is allocated and used to fetch block B. After this block is fetched, the contents of the victim line and FRC_0 are swapped. Then, the invalidation and eviction of block A are performed from FRC_0, which becomes free when the eviction is completed. In this way, fetch actions can be performed as long as there are free FRC entries (e.g. the fetch of block C can start in parallel at cycle 90). To ensure that there are free FRC entries, they are recycled. Thus, after block A

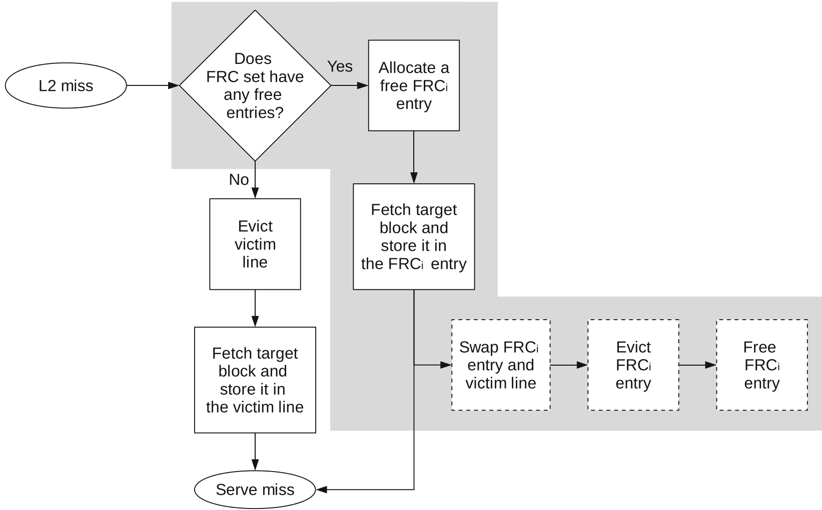

Fig. 3. Block diagram with the steps followed on an L2 miss. Those steps introduced with the FRC are highlighted in gray color.

has been replaced, FRC_0 is freed, which allows this entry to be used later by *Req. D.*

The swap operation guarantees that the victim line is never in a transient state (note that it is not represented with dashed lines in Fig. 2b), and that the invalidation and eviction of its contents are performed after the requested block is fetched. Consequently, FRC supports a higher cache level parallelism that allows responding to several requests at the same time. Furthermore, compared to the conventional approach, the lifetime of the victim block becomes longer when FRC is used.

Tags and control bits of blocks in transient state are stored in the FRC. Thus, to reduce tag lookup overhead, FRC is organized as a conventional cache, although its geometry (i.e. associativity and number of sets) can be different from that of the L2 cache. L2 accesses must search the requested block both in the target L2 bank and its associated FRC. A hit in the L2 bank is performed as in the conventional approach, while a hit in FRC for a block being fetched is enqueued until the fetch operation completes.

As shown in Fig. 3, the FRC approach modifies the classical miss management by adding the events highlighted in gray color. On an L2 miss (both in the L2 bank and the FRC), and if there are free entries in the FRC's set mapped to the missing block, the block is assigned to a FRC 's entry and the access is immediately propagated to the lower memory hierarchy level (early fetch). Once the fetch has been performed, the miss can be already served. In this way, the victim block eviction is taken out of the critical path. To manage the eviction without leaving L2 cache lines in a transient state, the data stored in the FRC's entry and the victim line are swapped. Thereby, the eviction is done from the

FRC's entry. Once the eviction has finished, the FRC's entry is set as free to handle subsequent L2 misses.

Finally, note that in case there is not any free entry in the FRC's set targeted by the missing block, the proposed approach operates like the conventional approach. In addition, FRC does not change the state of blocks stored in the cache, but only modifies the resources they are using. Thus, it does not affect the coherence protocol.

Overall, as experimental results will show, FRC has three main impacts on performance: (i) new requests do not wait (or wait much less) for cache block's evictions, which reduces the memory access latency, (ii) the lifetime of an L2 block becomes longer, decreasing the number of misses, and (iii) a higher MLP is achieved, since FRC allows immediate access to lower memory levels as long as there are free FRC entries.

5 Experimental Evaluation

To evaluate the proposal, we have modeled the FRC approach with the Multi2Sim [4] simulation framework. We focus on the Southern Islands GPU architecture from AMD, which is one of the most recent GPU architectures modeled on a detailed simulation framework. In particular, we model the characteristics of an HD7770 GPU [6], including CUs, L1 and L2 caches, memory controllers, and GDDR5 memory [7]. The L2 cache consists of two 16-way 128 KB banks, which is our baseline configuration. In addition, to evaluate the impact on performance of cache associativity and capacity, we evaluate two additional conventional L2 caches consisting of two 32-way 256 KB banks and two 32-way 512 KB banks. Both configurations are compared to the FRC one, which is composed of the baseline configuration plus two additional FRCs (1 per bank). We analyze the sensitivity our proposal to the number of FRC entries, which ranges between 4 and 512. All the evaluated FRC configurations, except the smallest one with 4 entries, are organized with 8-way sets.

Notice that the FRC approach represents a minor area increase over the baseline, since the area occupied by an additional FRC is much smaller than doubling or quadrupling the cache bank capacity, which would present roughly the same cost in area as adding 2048 and 6144 entries, respectively. Nevertheless, we conservatively assume that all the analyzed L2 cache configurations have the same access time.

For evaluation purposes, a subset of the OpenCL SDK 2.5 benchmarks [8] has been used, covering all the possible performance behaviors from the entire benchmark suite. These benchmarks are executed until completion.

5.1 Performance Analysis

System performance has been quantified in terms of Operations Per Cycle (OPC), which is analogous to its counterpart IPC used to evaluate CPU processors [7]. This metric accounts for the number of single scalar operations each

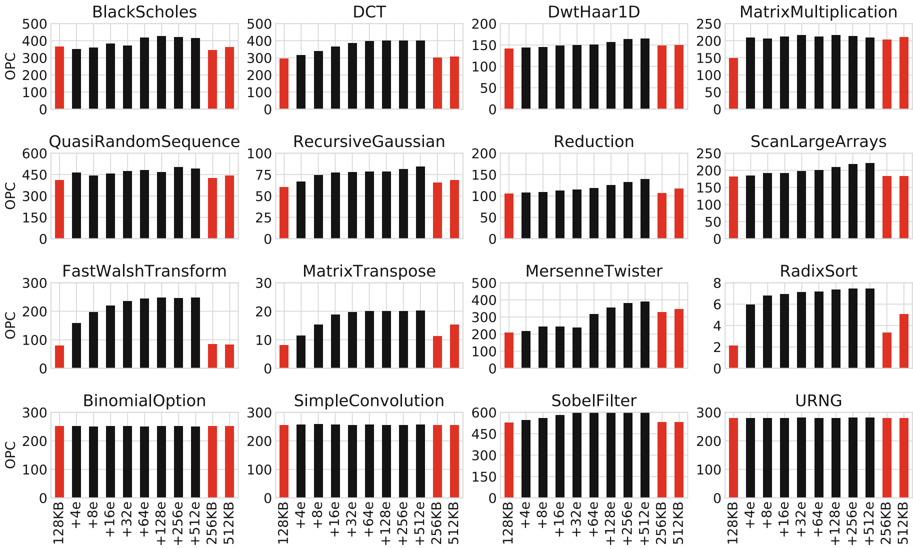

Fig. 4. Operations Per Cycle (OPC) across the studied applications. (Color figure online)

GPU instruction executes during the workload execution. For instance, if a given vector instruction is internally executed as 64 individual scalar operations, this metric accounts for 64 operations instead of only one instruction.

Figure 4 shows the OPC for the studied benchmarks. The red bar on the left side of each plot represents the 2 × 128 KB L2 baseline cache, and the two red bars on the right side represent the 2 × 256 KB L2 cache and the 2 × 512 KB L2 cache, respectively. The black bars show results of the FRC configuration varying the number of entries per FRC ranging from 4 to 512, labeled as *+Ne*, where N indicates the number of entries. The proposed approach achieves, across most of the studied applications, OPC improvements higher than 25% compared to the baseline, reaching improvements up to 150% in applications such as `FastWalshTransform` and `MersenneTwister`. In general, it can be observed that almost all the applications achieve their highest OPC with around 32 or 64 entries, which represents by 64× and 32× less area, respectively, than doubling the cache bank size to 256 KB. Moreover, in most applications, the performance achieved by FRC is much higher than that obtained by blindly increasing the L2 cache capacity with a higher associativity degree.

Three main behaviors can be appreciated:

- Smooth OPC increase. The OPC of applications exhibiting this behavior, which is the common one, increases in small steps with additional FRC entries until a given saturation point. This is the case of benchmarks such as `FastWalshTransform`, `MersenneTwister`, and `DCT`.
- Sharp OPC increase. Applications presenting this behavior show significant performance increase with just 4 FRC entries, but no remarkable

OPC improvement is observed with additional entries. This is the case of `MatrixMultiplication`.

– Similar OPC. Applications in this category experience the same performance across all the studied cache approaches. This is the case of `BinomialOption` and `URNG`, mainly due to their low number of memory accesses as discussed below. Obviously, the OPC of this type of applications is also not affected when enlarging the L2 cache size and associativity.

5.2 Analysis of Memory Subsystem Metrics

To provide insights into the OPC trend shown by the studied applications, we analyze the following metrics: number of misses measured in *Misses Per Kilo-Operation* (MPKO), percentage of misses served by FRC additional entries, and the L2 miss latency penalty.

Misses Per Kilo-Operation. We define the metric MPKO for GPUs with analogous meaning to the MPKI (Misses Per Kilo-Instruction), widely used when studying the cache hierarchy of the CPU counterparts. Figure 5 plots the results. It can be observed that the baseline configuration shows high MPKO values, which can be notably reduced by adding FRC entries. This fact confirms the benefits on performance brought by the FRC approach by keeping victim blocks in a non-transient state until fetch actions are completed. As a consequence, the hit ratio is improved compared to the conventional approach.

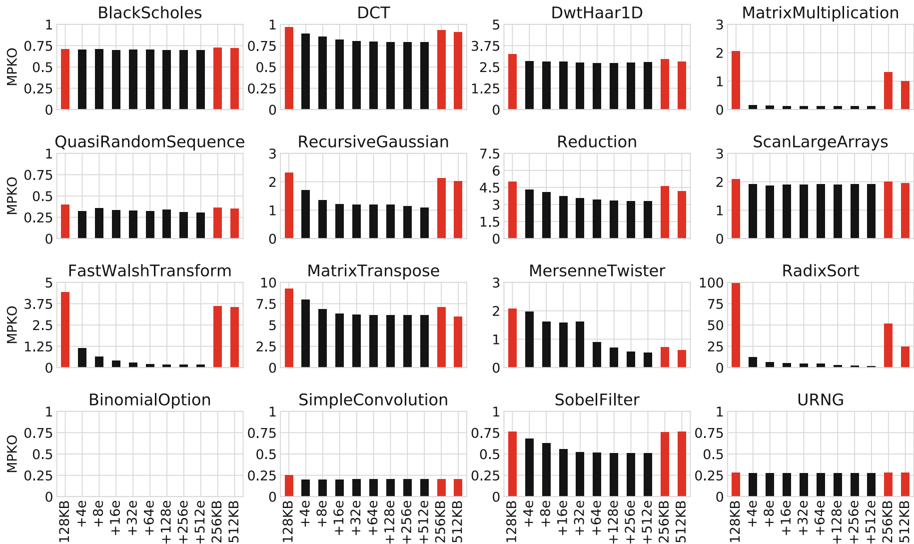

Fig. 5. Misses Per Kilo-Operation (MPKO) in the L2 cache.

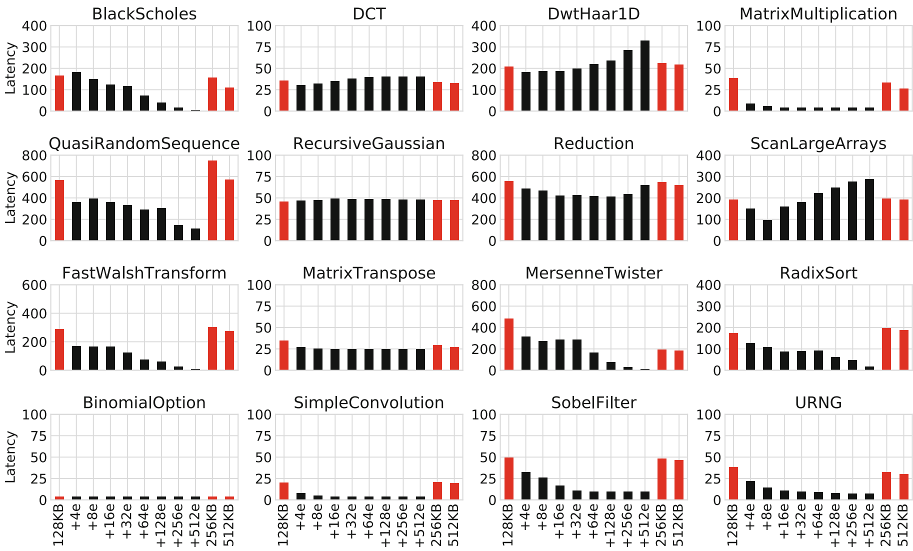

Fig. 6. Average L2 miss delaying latency quantified in processor cycles.

Overall, a clear inverse correlation between OPC and MPKO can be appreciated. However, in a few applications like `DwtHaar1D` and `Reduction`, a significant MPKO reduction over the baseline with a few FRC entries has a minimal effect on OPC. On the other hand, as observed, `BinomialOption` and `URNG` present a near-zero MPKO, meaning that no OPC gains can be achieved in these applications by acting on the L2 cache. However, there are applications like `BlackScholes`, `DCT`, `QuasiRandomSequence`, and `SobelFilter`, with a relatively low MPKO (below 1.5) in the baseline which improve their OPC with an FRC. In order to explain these behaviors, the MLP and memory latency are analyzed below.

L2 Miss Latency. L2 cache misses can be handled either by normal cache entries or by FRC entries. Misses handled by FRC entries can be considered as *fast* L2 misses since, as explained in Sect. 4, they are able to access to main memory with a minimum delay. In other words, the more misses handled by FRC entries the better the performance. Figure 6 plots the results of the L2 miss latency (excluding the actual main memory access time), quantified in processor cycles.

The use of FRC entries reduce the average L2 miss latency for almost all the applications. As observed, with just 4 FRC entries, latency is largely reduced with respect to the 256 KB and 512 KB cache configurations. In fact, the largest FRC configuration completely reduces the L2 contention in most benchmarks. Nevertheless, it can be seen that just 4 FRC entries only provide a slight latency improvement in some applications, thus large-sized FRCs are preferred. However,

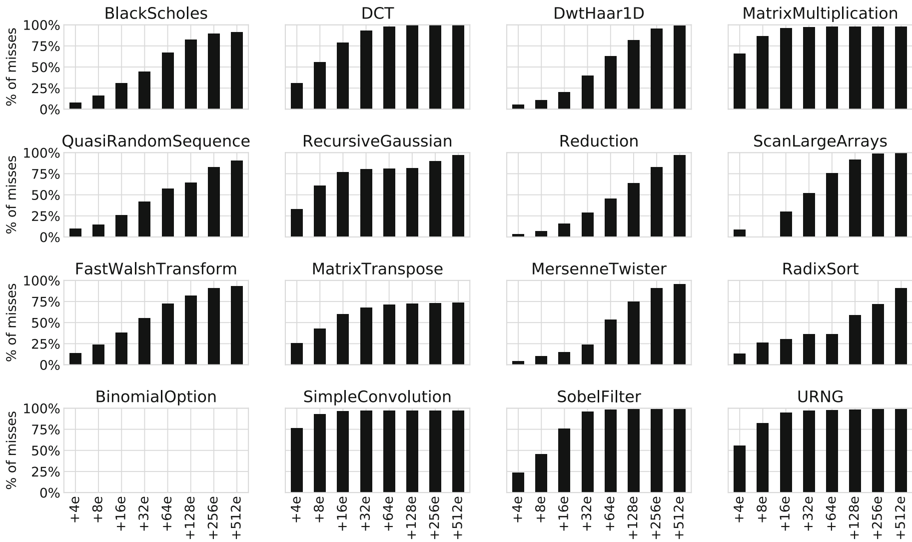

Fig. 7. Percentage of L2 misses handled by FRC entries.

`DwtHaar1D` and `ScanLargeArrays` suffer an increase in latency as the number of FRC entries grows over around 8 entries. This is because the parallelism level is higher than the baseline, which increases the memory contention. Notice that, in spite of this increase, the higher MLP turns into OPC improvements.

Percentage of Misses Served by the FRC. Since the service of misses is not stalled in case of consecutive replacements over the same victim line, MLP is also improved. Figure 7 shows the percentage of misses served by the FRC. As observed, FRC with only 64 entries handles by 75% of misses in most applications. Moreover, this percentage significantly rises, even to almost 100% in some benchmarks, for configurations smaller than the $+512e$ configuration.

The applications `Matrixtranspose` and `BinomialOption` show an unexpected behavior as the percentage of misses handled by FRC entries saturate in a relatively low number of entries, that is, this percentage does not increase even if more entries are added. In other words, the L2 cache misses are mostly handled by the cache itself instead of by FRC entries. This is due to two different reasons. First, the kernel of `Matrixtranspose` presents bursts of accesses targeting the same FRC set. This behavior can be improved by increasing FRC associativity (8-way in these experiments). Second, `BinomialOption` makes important use of the local memory of the CU, which significantly reduces the number of accesses to main memory.

6 Related Work

The GPU memory subsystem performance has been widely analyzed in recent years from different angles, including memory scheduling strategies [9–11], cache bypassing techniques [12,13], and optimizing the memory subsystem design [14–18]. This section summarizes prior work in this regard.

Elastic-Cache [14] supports fine-grained L1 cache line management for those kernels with irregular memory access patterns that do not efficiently exploit cache space. Auxiliary tags for fine-grained cache line management are stored in unused shared memory space, which is not fully occupied in many kernels.

Gebhart et al. [15] propose to dynamically adjust the storage partitioning among registers, primary caches, and scratchpads depending on the kernel memory requirements, resulting in a reduction of the on-chip access latencies.

IBOM [16] is an integrated architecture that leverages unused register file entries with lightweight ISA support to enlarge the L1 cache size. With enough cache capacity, a set balancing technique exploits underutilized sets to improve cache usage.

Other works have proposed additional memory structures to improve GPU performance. Wang et al. [17] incorporate a victim cache between L1 and L2 that presents the same capacity and associativity as the L1 cache. Reused blocks are kept in the L1 cache by enabling swap operations with the victim cache. Since a victim cache so large would impact on energy and area, unused entries from the register file and shared memory are proposed as an alternative to holding data that otherwise would remain in the victim cache.

In [18], the authors propose to allocate *TinyCaches* between each lane in a CU and the L1 cache to filter out memory requests to lower memory levels for energy saving purposes. By leveraging intrinsic characteristics of CUDA and OpenCL programming models, these caches are kept non-coherent to avoid incurring additional overheads.

All the above works primarily focus on L1 caches. In contrast, our proposed FRC design targets LLCs where all accesses from L1 are merged and contention greatly limits MLP. Furthermore, the FRC approach can be easily implemented in different memory subsystem architectures, since it does not change the actions required to handle misses, but the locations where these actions are performed (i.e. FRC entries).

7 Conclusions

This paper has presented a novel GPU cache subsystem design that leverages a small Fetch and Replacement Cache (FRC) between the Last-Level Cache (LLC) and the main memory. The design provides additional cache lines that allow prioritizing the fetch of incoming LLC cache blocks over the replacement of victim blocks. The proposed design boosts the system performance by increasing the Memory-Level Parallelism (MLP) and enlarging the lifetime of the victimized blocks.

FRC attacks by design three main cache performance related events, which results in a much better L2 cache management: (i) it reduces the number of Misses Per Kilo-Operation (MPKO) by keeping victim blocks in cache until fetch actions are completed, (ii) it reduces the miss latency by starting the fetch actions from main memory as soon as a miss rises, and (iii) it increases the MLP by unclogging new block requests whose victim line is already being replaced.

Experimental results have shown that, compared to a conventional LLC design, FRC increases the Operations Per Cycle (OPC) over 25% in all the applications suffering contention in main memory.

Acknowledgments. This work was supported by the Spanish *Ministerio de Economía y Competitividad* (MINECO) and Plan E funds under Grant TIN2015-66972-C5-1-R and TIN2016-76635-C2-1-R (AEI/FEDER, UE), and by the *Programa de Ayudas de Investigación y Desarrollo* (PAID) *de la Universitat Politècnica de València*.

References

1. Huang, S., Xiao, S., Feng, W.: On the energy efficiency of graphics processing units for scientific computing. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8 (2009)
2. Glenis, A., Petridis, S.: Performance and energy characterization of high-performance low-cost cornerness detection on GPUs and multicores. In: Proceedings of the 5th International Conference on Information, Intelligence, Systems and Applications, pp. 181–186 (2014)
3. Top500.org: Top500 Supercomputer Sites. http://top500.org
4. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2Sim: a simulation framework for CPU-GPU computing. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 335–344 (2012)
5. AMD: Southern Islands Series Instruction Set Architecture (2012)
6. AMD: AMD Graphics Cores Next (GCN) Architecture White Paper (2012)
7. Candel, F., Petit, S., Sahuquillo, J., Duato, J.: Accurately modeling the on-chip and off-chip GPU memory subsystem. Future Gener. Comput. Syst. **82**, 510–519 (2018)
8. AMD: AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) (2012)
9. Mu, S., Deng, Y., Chen, Y., Li, H., Pan, J., Zhang, W., Wang, Z.: Orchestrating cache management and memory scheduling for GPGPU applications. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **22**(8), 1803–1814 (2014)
10. Jia, W., Shaw, K.A., Martonosi, M.: MRPB: memory request prioritization for massively parallel processors. In: Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture, pp. 272–283 (2014)
11. Sethia, A., Jamshidi, D.A., Mahlke, S.: Mascar: speeding up GPU warps by reducing memory pitstops. In: Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture, pp. 174–185 (2015)
12. Li, C., Song, S.L., Dai, H., Sidelnik, A., Hari, S.K.S., Zhou, H.: Locality-driven dynamic GPU cache bypassing. In: Proceedings of the 29th International ACM Conference on Supercomputing, pp. 67–77 (2015)

13. Liang, Y., Xie, X., Wang, Y., Sun, G., Wang, T.: Optimizing cache bypassing and warp scheduling for GPUs. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2018, to appear)
14. Li, B., Sun, J., Annavaram, M., Kim, N.S.: Elastic-cache: GPU cache architecture for efficient fine- and coarse-grained cache-line management. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, pp. 82–91 (2017)
15. Gebhart, M., Keckler, S.W., Khailany, B., Krashinsky, R., Dally, W.J.: Unifying primary cache, scratch, and register file memories in a throughput processor. In: Proceedings of the IEEE/ACM 45th Annual International Symposium on Microarchitecture, pp. 96–106 (2012)
16. Mu, S., Deng, Y., Chen, Y., Li, H., Pan, J., Zhang, W., Wang, Z.: IBOM: an integrated and balanced on-chip memory for high performance GPGPUs. IEEE Trans. Parallel Distrib. Syst. **29**(3), 586–599 (2018)
17. Wang, J., Fan, F., Jiang, L., Liang, X., Jing, N.: Incorporating selective victim cache into GPGPU for high-performance computing. Wiley Concurr. Comput.: Pract. Exp. **29**(24), 1–11 (2017)
18. Sankaranarayanan, A., Ardestani, E.K., Briz, J.L., Renau, J.: An energy efficient GPGPU memory hierarchy with tiny incoherent caches. In: Proceedings of the International Symposium on Low Power Electronics and Design, pp. 9–14 (2013)