Efficient Management of Cache Accesses to Boost GPGPU Memory Subsystem Performance

Francisco Candel, Alejandro Valero, Salvador Petit, *Member, IEEE,* and Julio Sahuquillo, *Member, IEEE*

Abstract—To support the massive amount of memory accesses that GPGPU applications generate, GPU memory hierarchies are becoming more and more complex, and the Last Level Cache (LLC) size considerably increases each GPU generation. This paper shows that counter-intuitively, enlarging the LLC brings marginal performance gains in most applications. In other words, increasing the LLC size does not scale neither in performance nor energy consumption. We examine how LLC misses are managed in typical GPUs, and we find that in most cases the way LLC misses are managed are precisely the main performance limiter. This paper proposes a novel approach that addresses this shortcoming by leveraging a tiny additional Fetch and Replacement Cache-like structure (FRC) that stores control and coherence information of the incoming blocks until they are fetched from main memory. Then, the fetched blocks are swapped with the victim blocks (i.e., selected to be replaced) in the LLC, and the eviction of such victim blocks is performed from the FRC. This approach improves performance due to three main reasons: i) the lifetime of blocks being replaced is enlarged, ii) the main memory path is unclogged on long bursts of LLC misses, and iii) the average LLC miss latency is reduced. The proposal improves the LLC hit ratio, memory-level parallelism, and reduces the miss latency compared to much larger conventional caches. Moreover, this is achieved with reduced energy consumption and with much less area requirements. Experimental results show that the proposed FRC cache scales in performance with the number of GPU compute units and the LLC size, since, depending on the FRC size, performance improves ranging from 30% to 67% for a modern baseline GPU card, and from 32% to 118% for a larger GPU. In addition, energy consumption is reduced on average from 49% to 57% for the larger GPU. These benefits come with a small area increase (by 7.3%) over the LLC baseline.

Index Terms—GPU, memory hierarchy, miss management.

1 Introduction

Nowadays, GPU (Graphics Processing Unit) architectures have acquired a great relevance in the high-performance computing field. One of the main reasons has been that GPUs are energetically more efficient [11], [12] when running massively parallel applications, since they provide a much higher level of parallelism than their CPU counterparts with a much better performance to power ratio. In fact, many of the current most powerful and energy-efficient supercomputers, ranked in both the Top500 and Green500 lists [37], rely on GPUs.

GPU architectures are optimized to run applications composed of thousands of logical threads. Given that these applications demand an ever-increasing amount of computational and memory resources, successive GPU architectures include more *multiprocessors* (i.e., compute units) and a larger on-chip memory subsystem. For instance, NVIDIA has continuously enlarged the Last-Level Cache (LLC) size in 2MB on recent architectures (e.g., LLC sizes of Maxwell [26], Pascal [27], and Volta [14] GPUs are 2MB, 4MB, and 6MB, respectively). Coupling GPUs with larger memory subsystems enables a higher Memory-Level Parallelism (MLP). However, because of the poor data temporal locality of GPU applications, upon a *fast and relatively very long* burst of LLC (i.e., L2) accesses it is likely that a significant number of accesses miss in the cache, requiring access to the off-chip memory, which can severely hurt the system performance.

A straightforward solution consists of drastically increasing the L2 cache size with the aim of accommodating the entire working set of the application on chip. Unfortunately, enlarging the L2 cache not only brings much lower performance gains in GPUs than in CPUs [9], [19], but also translates into a high area overhead as well as a huge static energy consumption, which aggravates as transistor size shrinks [15].

In this paper, we look into the reasons explaining the poor performance gains of GPU memory subsystems, and we find that a key aspect is the way L2 cache misses are managed in typical caches. In particular, a typical cache miss management gets clogged on fast, long bursts of cache misses, which increases memory latencies and limits MLP. Moreover, the lifetime of memory blocks is shortened, rising the amount of memory misses even for those applications with low temporal locality and low cache hit ratio.

The previous rationale means that the L2 cache management is a key design concern that should be tackled to improve the GPU performance. This paper proposes an energy-efficient L2 cache design aimed at boosting MLP by adding a tiny Fetch and Replacement Cache-like structure (FRC) that provides additional reusable cache lines that help unclog the memory subsystem. The proposed approach prioritizes the fetch of incoming L2 cache requests and delays the eviction of the blocks, which helps alleviate memory latencies and improve the cache hit ratio.

The proposal has been modeled and evaluated in both AMD Polaris [3] and Vega [4] GPU architectures, although the results would apply to most of the current GPU architectures provided that they implement a similar memory subsystem and organization. For instance, some NVIDIA L2 caches use a replacement algorithm other than LRU and a 32B line size [21]. However, these characteristics are orthogonal to our proposal. In particular, experiments consider two Polaris GPU cards, namely RX540 and RX570, with a different number of compute units and L2 cache sizes to show the

• *F. Candel, S. Petit, and J. Sahuquillo are with the Department of Computer Engineering, Universitat Politècnica de València, Spain. E-mails: fracanma@inf.upv.es, {spetit, jsahuqui}@disca.upv.es.*

• *A. Valero is with the Departmento de Informática e Ingeniería de Sistemas, Instituto Universitario de Ingeniería de Aragón, Universidad de Zaragoza, Spain. E-mail: alvabre@unizar.es.*

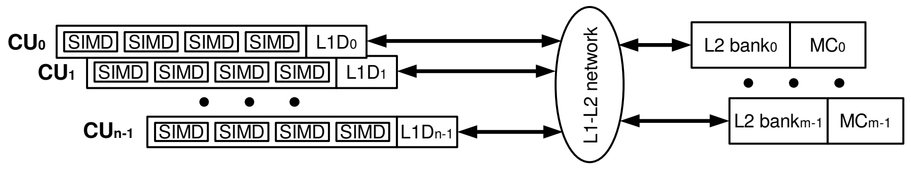

Fig. 1. Diagram of an AMD Polaris GPU.

scalability of FRC in terms of performance and energy, whereas a Vega64 card is also studied to show how the proposed FRC approach behaves with an enhanced memory subsystem using HBM technology and a higher clock frequency.

The proposal has been modeled using the state-of-the-art Multi2Sim [38] and CACTI [35] simulation frameworks, which are a cycle-accurate GPU simulator and an analytical model for both on-chip and off-chip memories, respectively, both widely used in the academia and the industry. Experimental results show that, compared to a conventional design, FRC improves the average system performance (OPC) of the RX540 between 30% and 67% depending on the FRC size, whereas these percentages rise up to 32% and 118%, respectively, for the larger RX570 GPU. Moreover, in most applications, the performance achieved by adding a small FRC is much higher than simply increasing the L2 cache capacity or associativity. In addition, compared to the conventional approach, energy savings fall in between 49% and 57% for the RX570 GPU. These benefits come with a small L2 cache area increase by 7.3% over the baseline. Finally, in spite of an improved memory subsystem with the Vega64 GPU, the FRC approach still boosts the average OPC from 16% to 54%.

This paper extends the work in [6] in four main ways: i) a hardware implementation for FRC is presented, ii) FRC has been modeled and evaluated on the recent AMD Polaris and Vega GPU architectures, iii) performance scalability has been studied by analyzing how FRC behaves with an increasing number of compute units and L2 cache sizes, and iv) energy consumption and area results are discussed.

The remainder of this work is organized as follows. Section 2 describes the architecture of the AMD Polaris family of GPUs. Section 3 motivates this work. In Section 4, the proposed approach is introduced. Section 5 presents the experimental results. Section 6 summarizes related studies about GPU memory subsystems. Finally, Section 7 summarizes the paper.

2 Background of GPU Architectures

This section provides some background of the architecture of modern GPUs. Since this paper primarily uses the AMD Polaris family of GPUs as a driving example, the AMD terminology is used throughout this work.

Figure 1 depicts a block diagram of an AMD Polaris GPU. This GPU includes up to 36 Compute Units (CUs), each one implementing the 4th version of the *Graphics Core Next* (GCN) [2] 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 the LLC. L2 banks contain interleaved block addresses at a granularity of 256 bytes, and each bank is connected to a dual-channel memory controller (MC) 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

3.1 Conventional Cache Miss Management

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 executing in parallel causes sudden bursts of memory accesses, which involve a high number of cache replacements. As a consequence, in a relatively short interval of time, a relatively high number of cache lines 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 each cache line, the management of cache replacements becomes a major performance bottleneck, which can heavily increase memory latencies and reduce the MLP and the L2 hit ratio.

To help understand the problem, Figure 2 plots a time diagram with the events involved in three consecutive replacements, all of them targeting the same L2 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 Figure 2a, which shows the behavior of a conventional replacement approach, *Req. B* triggers the replacement of the currently stored block (block *A*). From this point forward, the line storing the victim block is in a transient state (represented in 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 should be

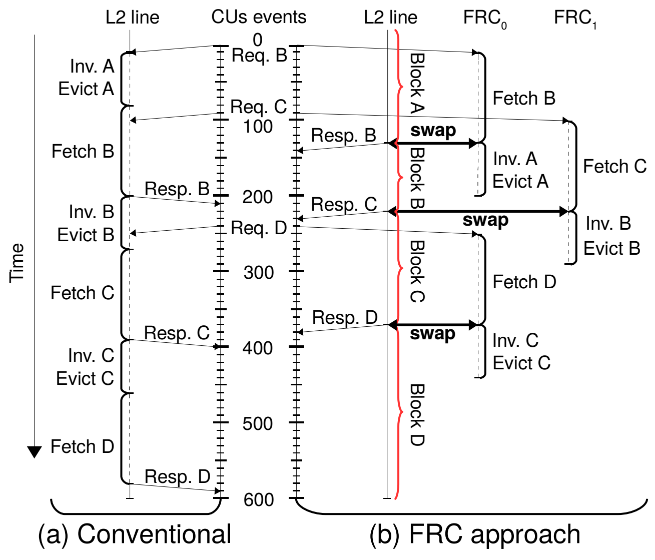

Fig. 2. Sequence of events involved in three consecutive replacements targeting the same L2 cache line for both the conventional and the proposed approaches.

performed. Once the line is released, the requested incoming block (B), must be fetched from main memory and written in that line.

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

Delaying requests increases memory latencies and reduces MLP. Moreover, the hit ratio is also reduced, since i) the invalidation and eviction of the victim block 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 memory instruction accessing to block A would only hit between cycles 0 and 90, and would miss afterward.

3.2 A Novel Cache Miss Management Approach

The proposed approach is aimed at improving MLP and LLC hit ratio while reducing miss latencies. With these aims, we implement a Fetch and Replacement Cache (FRC) in each L2 cache bank. The FRC provides additional cache lines that allow to i) start fetching from memory as soon as an L2 miss is detected, which reduces the miss latency and increases the MLP, and ii) delaying invalidation and eviction actions *until* the requested block is fetched, which enlarges the lifetime of victim blocks and the overall hit ratio.

Figure 2b shows how the FRC can help improve the management of consecutive replacements in the same line. By cycle 10, when *Req. B* misses in the L2 cache, instead of invalidating the victim block (i.e., block A), a free FRC entry (FRC_0) is allocated and used to fetch block B. After this block is fetched, the contents of the line storing block A and FRC_0 are swapped. Then, the invalidation and eviction of block A are performed from FRC_0, which is freed when the eviction is completed. In this way, fetch actions can be immediately start 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, once block A is replaced, FRC_0 is freed, which allows this entry to be used later by *Req. D*. Recycling entries allows FRC to be smaller and thus more efficient than conventional approaches regarding energy consumption and area overhead.

The swap operation guarantees that the line storing the victim block is never in a transient state (note the lack of dashed lines in the plot below the L2 line of Figure 2b), and that the invalidation and eviction of the victim block 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 is enlarged when FRC is used.

Overall, as experimental results will show, the FRC has three main positive impacts on performance: i) reduces the memory access latency, ii) enlarges the lifetime of L2 cache blocks, and iii) exploits a higher MLP.

3.3 Potential FRC Performance Benefits

This section explores the potential performance benefits of the FRC approach and where they come from. To this end, the proposal is compared against two approaches, a fully-associative (FA) L2 cache and an FA L2 cache working together with a victim cache (FA+VC). The FA scheme is sized with the same number of entries as our experimental baseline (see Section 5 for further experimental details) and it is used to check the benefits coming from reducing the conflict misses. Notice that FA imposes an upper-bound for performance with respect to alternative set mapping strategies [25]. On the other hand, the FA+VC scheme is chosen to compare the potential benefits of a victim cache compared to our approach. In order to explore the potential performance, experiments assume that the additional structures (both VC and FRC) have an unbounded number of entries.

Performance is evaluated for the RX540 GPU in terms of Operations Per Cycle (OPC) and average number of execution cycles per wavefront in a kernel. The OPC is a performance metric analogous to the IPC, which is used to evaluate conventional processors [5]. An *operation* refers to the work performed by an individual thread when executing its corresponding part of a SIMD instruction. For instance, in our experimental platform, a SIMD unit can execute instructions from up to 64 threads, each one performing a scalar operation, which accounts for 64 operations. Regarding the execution cycles, they are split in two main categories referred to as compute cycles and memory cycles. The former indicates the mean time a wavefront is executing instructions and the latter the mean time a wavefront is blocked because it is waiting for a memory access.

For illustrative purposes, a pair of benchmarks showing two common and representative behaviors are presented (see Section 5). Figure 3 shows the results. As observed, for `DCT`, the FA cache

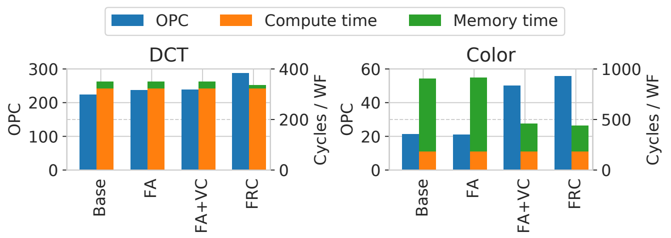

Fig. 3. Operations Per Cycle (left Y-axis) and average execution cycles split in compute and memory cycles (right Y-axis) for the studied approaches.

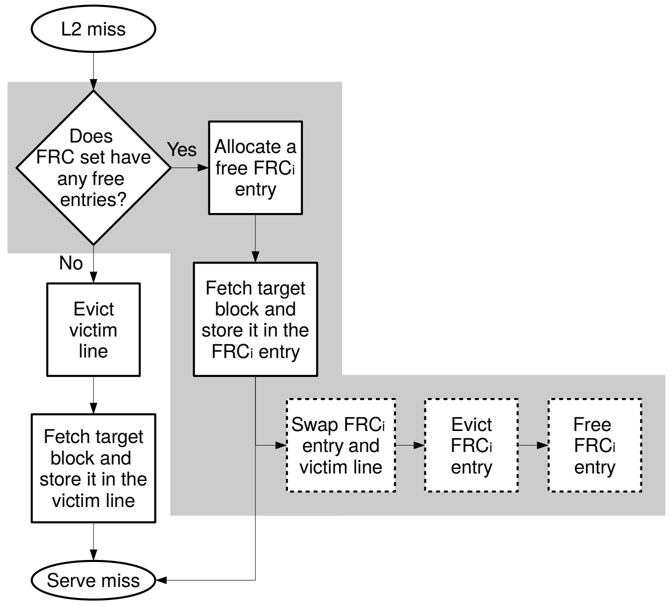
L2 miss
Does FRC set have any free entries?
Yes
Allocate a free FRCi entry
No
Evict victim line
Fetch target block and store it in the FRCi entry
Fetch target block and store it in the victim line
Swap FRCi entry and victim line
Evict FRCi entry
Free FRCi entry
Serve miss

Fig. 4. Block diagram with the steps followed on an L2 miss. Those steps introduced with the FRC are highlighted in gray color.

improves performance (i.e., OPC in the left Y-axis) by 6% over the baseline thanks to reducing the number of conflict misses. On the other hand, no performance gains can be observed in `Color`, where long bursts of cache accesses many times exceed the cache capacity, and capacity misses dominate over conflict misses. In this application, the OPC is improved by 137% over the baseline when adding the VC, which helps to reduce capacity misses; however, the VC slightly helps in `DCT` since capacity misses are not as critical as in `Color`. The main reason is that `Color` is a memory-intensive kernel, where memory cycles dominate over compute cycles. On the contrary, in `DCT`, the compute cycles dominate the execution time, hence, little can be done by enlarging the cache capacity with a VC.

To sum up, it can be concluded that to enlarge the L2 cache size and/or to increase its associativity either directly or indirectly (i.e., with an additional memory structure) can improve the performance in some (especially memory-bounded) applications but it cannot in some others (compute-bounded). However, looking at the FRC with the same number of entries as the FA+VC approach but with a different data management, the system performance is significantly boosted in both kernels (by 29% and 163% for `DCT` and `Color`, respectively). The main reason is that the primary aim of FRC is not only to reduce the number of either conflict or capacity misses but to improve the MLP and to further reduce the memory access latency. Notice too that, for the FRC, the average time a wavefront is blocked for memory is smaller with respect to the other approaches so that, taking into account all the wavefronts of the kernel together, it turns into significant performance gains.

4 FRC IMPLEMENTATION

Figure 4 illustrates a block diagram with the steps involved on an L2 cache miss. The highlighted steps in gray color correspond to the proposed FRC approach. Upon an L2 miss (both in the L2 bank and the FRC), and if there are free entries in the target FRC set, the block is assigned to an FRC entry and the access is forwarded to the lower memory hierarchy level. This process is referred to as an early fetch. Once the early fetch is performed, the missing data can be already delivered to the processor. 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 entry and the line storing the victim block (*victim line*) are swapped. Thereby, the eviction is handled from the FRC entry. Once the eviction finishes, the FRC entry is freed and enabled to handle subsequent L2 misses. In case that there is no free entry in the target FRC set, the FRC approach works like the conventional approach.

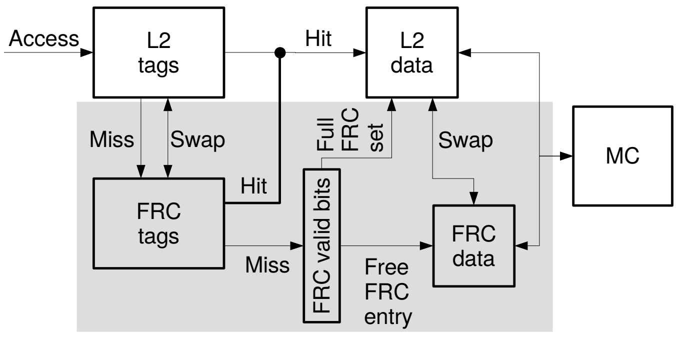

Fig. 5. FRC hardware block diagram.

Figure 5 depicts a hardware block diagram of the FRC approach. The main focus of this paper is not to deal with the optimal implementation but on providing some insights on the design. A refined design for enhanced performance is beyond the scope of this paper. The FRC is plotted within the gray box, and, similarly to the L2 cache, includes the FRC tag and data arrays. For illustrative purposes, the valid bits are plotted in a different box. The access to the L2 tags and the FRC tags are performed sequentially and this is the way modeled in the experimental results. In practice, however, these structures can be indexed with the target block address in parallel or within the same cycle to avoid any latency penalty.

On an L2 cache access, the tags of the target set are looked up on a first stage. On an successful tag comparison, the requested block is retrieved from the L2 data array on a second stage and the FRC is not used. Otherwise, the FRC tags are looked up. On a miss in both the L2 and FRC tag array, a free block entry in the target FRC set is sought. Depending on whether the FRC set has a free entry or not, the fetched block from main memory is written into the FRC or the L2, respectively. In the former case, once the fetch completes, the L2 victim block is swapped with the FRC block. On the other hand, that is, on a hit in the FRC tag array, the request is served by the L2 data array. In this case, the request waits until the swap operation completes.

Finally, note that the FRC approach does not affect the state of the cache blocks, thus, it does not affect the coherence protocol.

5 EXPERIMENTAL EVALUATION

The FRC approach has been modeled and evaluated with the Multi2Sim [38] simulation framework. The simulation results include performance metrics and cache memory statistics required to compute the overall energy consumption.

We focus on the AMD Polaris GPU architecture. The RX540 GPU [2] has been modeled, including CUs, L1 and L2 caches, memory controllers, and GDDR5 memory modules [5]. The RX540 consists of 8 CUs, each one implementing the 4th version of the GCN core. The L2 cache is composed of two 32-way 256KB banks, which has been used as the baseline configuration.

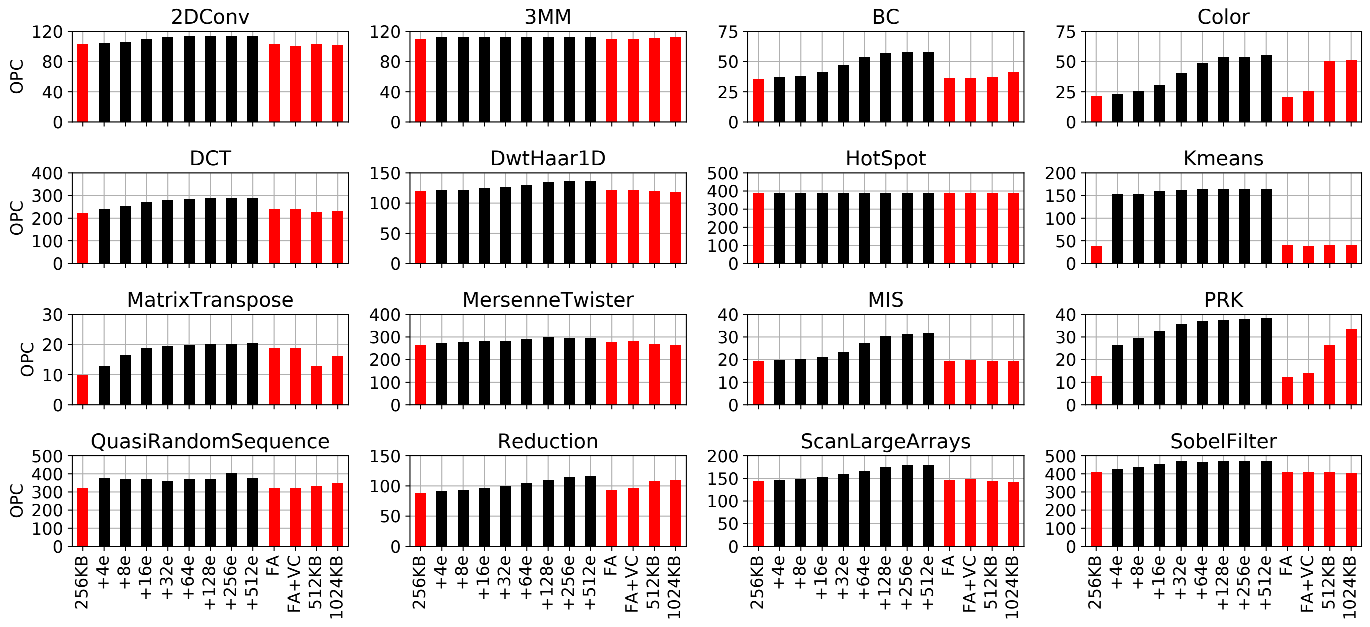

Fig. 6. Operations Per Cycle (OPC) of the RX540 across the studied applications.

The FRC consists of the L2 conventional cache plus two additional FRCs (one per bank). We analyze the performance sensitivity of our approach to the number of FRC entries, ranging from 4 (256B) to 512 (32KB) entries. All the evaluated configurations, except the smallest one, are organized as 8-way set-associative caches[1]. As mentioned above, the FRC is compared to the FA and FA+VC approaches. Experiments assume a fully-associative 32KB VC per bank, which matches the tested maximum FRC size. In addition, two conventional L2 caches consisting of two 32-way 512KB banks and two 32-way 1024KB banks are also evaluated. All the memory structures implement 64B lines.

To study the FRC performance scalability, we modeled the RX570 GPU with the same Polaris architecture as the RX540 GPU, but including up to 32 CUs and 8 L2 cache banks.

CACTI [35] has been used to estimate timing, energy, and area of the proposal. We have assumed a 32nm technology node and a 1.2GHz clock frequency. All the FRC caches are small enough to fit their access time within 1 clock cycle, whereas a swap operation is estimated to take 3 cycles to complete. Despite their fully-associative geometry, for comparison purposes, we conservatively assume the access time of the VCs to be the same as the FRCs. Regarding the L2 cache, it has been modeled with a 10-cycle access latency regardless of the cache geometry and capacity. The reader is referred to Section 5.4.1 for further experimental details about energy consumption.

Results have been obtained for 29 benchmarks from the OpenCL SDK [1], Rodinia [8], Pannotia [7], and PolyBench [29] benchmark suites. For illustrative purposes, a subset of 16 benchmarks are shown. All the benchmarks are run until completion.

5.1 System Performance Analysis

Figure 6 shows the OPC achieved by the RX540 GPU for the studied benchmarks. The red bar on the left side of each plot represents the 2×256KB L2 baseline cache, and the four red bars on the right side represent the 2×256KB FA L2 cache, the FA L2 cache with a 2×32KB VC (FA+VC), the 2×512KB L2 cache, and the 2×1024KB L2 cache, respectively. The black bars show results of the proposal varying the number of entries per FRC from 4 to 512, labeled as *+Ne*, where *N* indicates the number of entries.

The proposed approach achieves, across most of the studied applications (14 out of 16), OPC improvements higher than 10% over the baseline, reaching improvements up to 200% in applications such as `Kmeans` and `PRK`. It can be observed that OPC improves, in general, as the number of entries increases up to 64 or 128, where it saturates in most applications. In most of the benchmarks, the proposal performs better than blindly increasing the L2 cache associativity and capacity. Enlarging the cache capacity enhances the performance over a higher number of ways in benchmarks like `Color`, `PRK`, and `Reduction`.

According to the FRC performance, three behaviors can be appreciated:

- Smooth OPC increase. The OPC of applications exhibiting this behavior, which is the common case, gradually increases with additional FRC entries until a given saturation point, which is achieved with a small FRC of 64 or 128 entries. Examples are `DCT`, `MatrixTranspose`, and `ScanLargeArrays`.
- Sharp OPC increase. This behavior show a significant performance increase with just 4 FRC entries, but no remarkable OPC improvement is observed with additional entries. This is the case of `Kmeans`.
- Similar OPC. Applications in this category experience the same performance across all the studied cache approaches. This is the case of `3MM` and `HotSpot`, mainly due to their relatively low number of memory accesses, as shown in Section 5.2. Of course, the OPC of this type of applications is neither affected when either enlarging the cache associativity or capacity.

Overall, FRC boosts the OPC between 30% (*+4e*) and 67%

1. Higher associativity has been explored for enhanced performance. However, the marginal performance gains do not compensate the extra energy and area consumption. Therefore, all the presented results assume 8-way associativity.

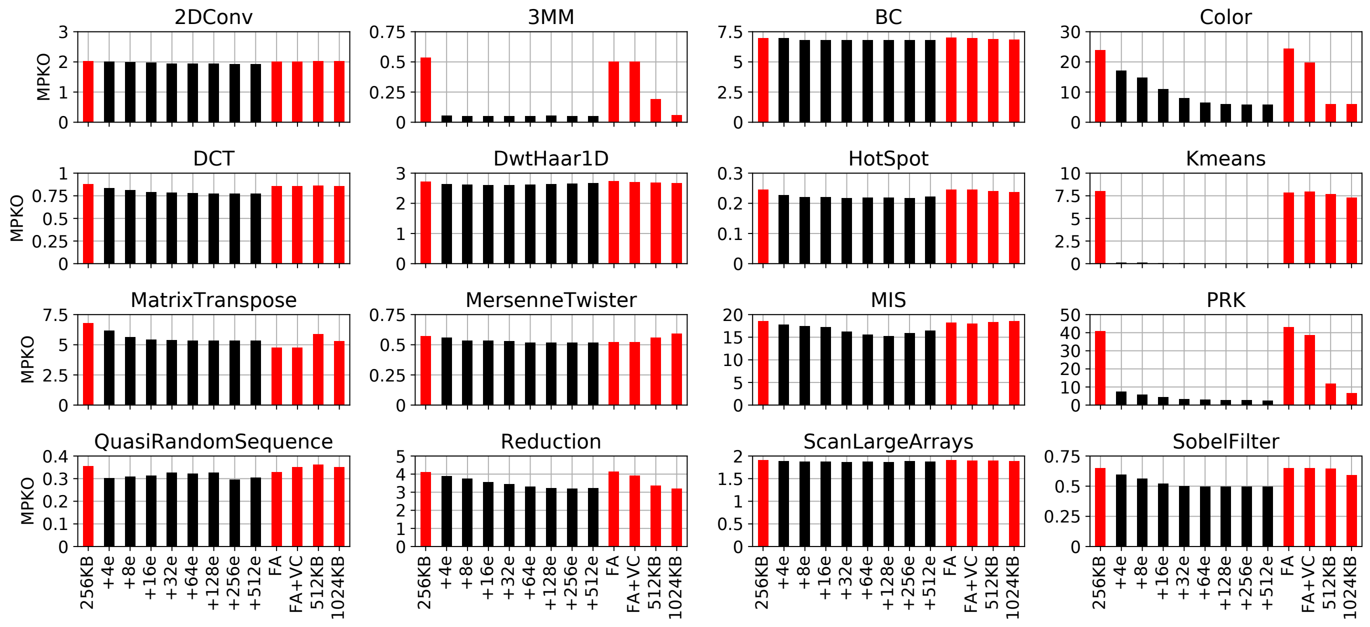

Fig. 7. Misses Per Kilo-Operation (MPKO) in the L2 cache of the RX540.

(*+512e*) on average compared to the baseline. These values drop to 20% and 27% for the 512KB and 1024KB caches, respectively, and they are less than 10% for the FA schemes.

5.2 Memory Subsystem Performance Analysis

To provide insights into the OPC trend shown by the RX540 GPU, this section evaluates the memory hierarchy performance.

5.2.1 Misses Per Kilo-Operation

Misses per Kilo-Operations (MPKO) can be defined with analogous meaning to the MPKI (Misses Per Kilo-Instruction), widely used to study the cache hierarchy of the CPU counterparts. Figure 7 depicts the results. By adding FRC entries, the MPKO is reduced on average between 23% (*+4e*) and 31% (*+512e*) compared to the baseline approach. Moreover, the MPKO can be completely or mostly eliminated in applications like `Kmeans` and `PRK`. Note that in both benchmarks, FA+VC provides significantly lower MPKO reductions than FRC. This is because FRC does not only enlarge the lifetime of victim blocks (as a victim cache does) but also because it keeps them in a non-transient state. As a consequence, the number of hits improves over the conventional approaches.

Overall, an inverse correlation between OPC and MPKO can be appreciated. For kernels with a near-zero MPKO, (e.g., below 1) one might expect that increasing the number of hits would not improve the OPC. Examples are `3MM` and `HotSpot`. However, there are applications with an MPKO lower than 1 like `DCT`, `QuasiRandomSequence`, `MersenneTwister`, and `SobelFilter`, which improve their OPC with FRC. In order to explain this behavior, memory latency and bandwidth consumption are analyzed below.

5.2.2 Memory Latency and Bandwidth Consumption

L2 cache misses can be handled by either normal cache entries or FRC entries, however, FRC handles misses *faster* than the L2 cache since, part of the main memory latency is hidden by moving eviction and invalidation actions out of the critical path. In other words, the higher the number of misses handled by FRC the better the performance.

Figure 8 plots the average L2 miss latency results (excluding the actual DRAM access time without contention), quantified in clock cycles. The miss latency is split in three main components according to its causes (see Figure 2): invalidations, evictions, and fetches (L2_to_MM). The former category is due to invalidating and writing back (if necessary) the L1 copies of the blocks that are evicted from L2. The eviction category accounts for the latency due to evicting L2 blocks and writing back their data to main memory. Finally, the fetch latency refers to the time fetching target blocks but excluding the actual DRAM access time. That is, its value is only affected by main memory contention.

The use of FRC entries reduces the average L2 miss latency in some applications. Especially, the latency caused by evictions is removed in most applications with 512 FRC entries. The figure shows that the FRC approach also improves performance due to latency reduction. In particular, `BC`, `MersenneTwister`, and `QuasiRandomSequence`, which do not benefit from MPKO improvement, present a significant reduction in memory latencies ranging from 84% to 93% over the baseline. In contrast, in some applications like `2DConv`, `DCT`, `DwtHaar1D`, `Reduction`, or `ScanLargeArrays` the miss latency increases, in spite of removing the eviction latency.

To provide insights on this increase, Figure 9 shows the traffic in bytes per cycle from the L2 cache to main memory and vice versa. It can be seen that the traffic rises in these applications with the number of FRC entries. This is because adding more entries enables a higher MLP. In turn, such an MLP increases memory contention and L2 to memory latency, but also improves OPC since the memory latency growth can be partially hidden by the GPU massive parallelism, while the higher MLP enhances the system throughput.

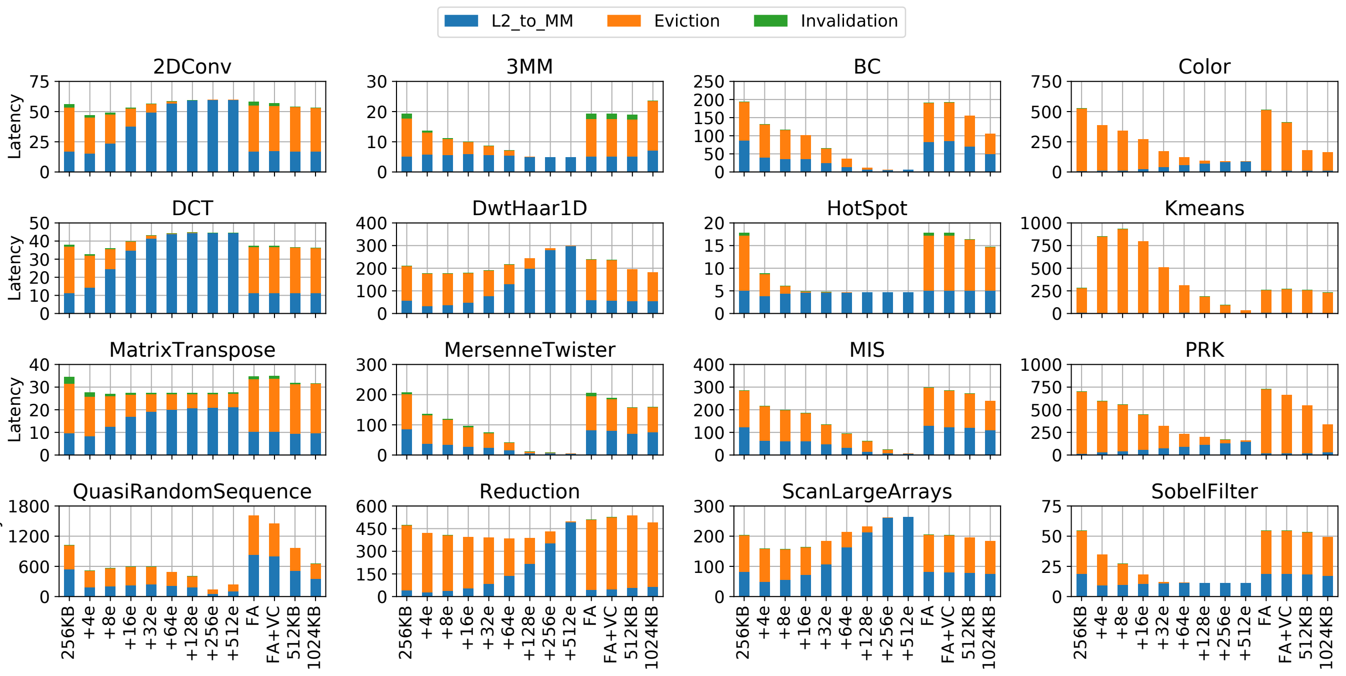

Fig. 8. Average L2 miss latency (excluding main memory access time) in processor cycles for the RX540.

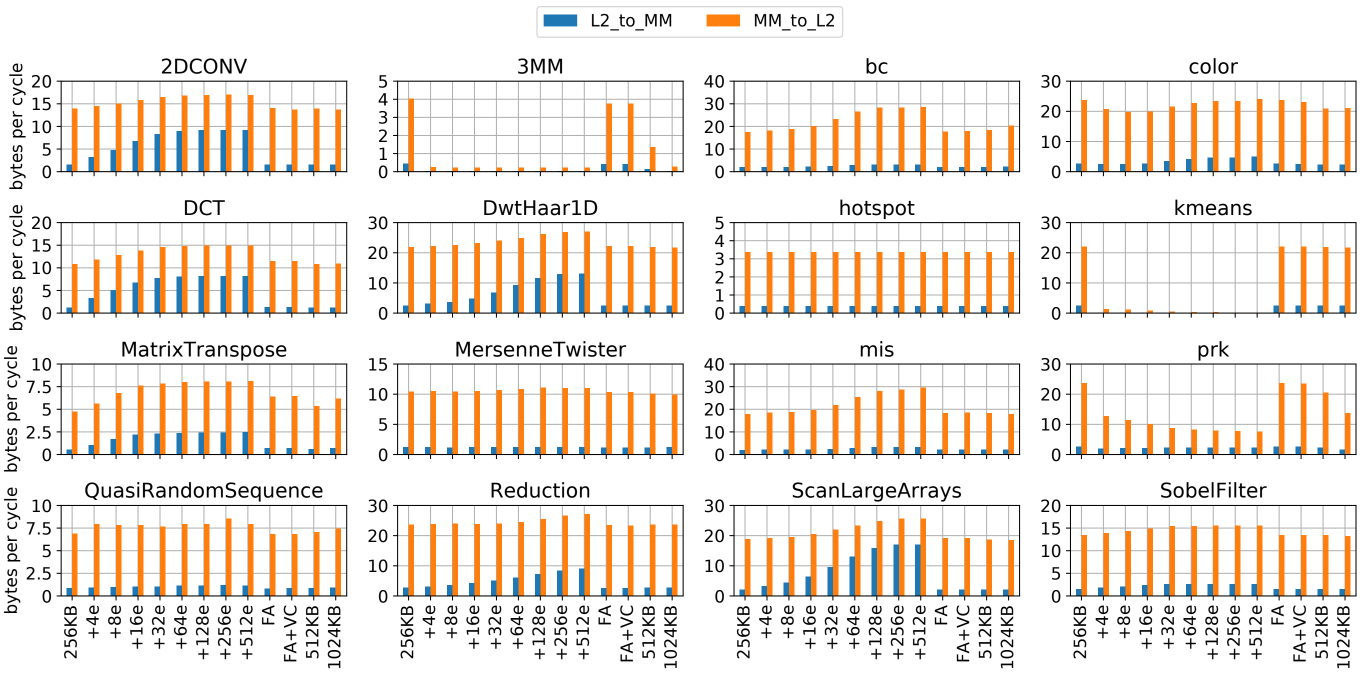

Fig. 9. Traffic from the L2 to main memory and vice versa on the RX540.

5.3 Impact on Performance of Increasing the Number of Compute Units

The previous sections have focused on the performance of the RX540 GPU consisting of 8 CUs and 2 L2 cache banks. FRC, however, is expected to behave better compared to the other approaches with additional computational power and memory subsystem capabilities, since a higher L2 contention is expected. This section studies the FRC performance in the RX570 GPU, which implements 4× more compute units (32 CUs) and L2 cache banks (8 banks).

Figure 10 presents the OPC results. As expected, this GPU outperforms the smaller RX540 GPU. Results also show that, compared to the same baseline, OPC improvements of the proposal are higher than those achieved by the RX540 GPU. In most benchmarks, OPC improvements range from 40% to 300%, whereas these percentages fall down to 10% and 200%, respectively, for the RX540 GPU. To sum up, these results point out that the FRC approach performs even better with a larger L2 and potentially higher memory level parallelism.

Overall, FRC improves OPC between 32% (*+4e*) and 118%

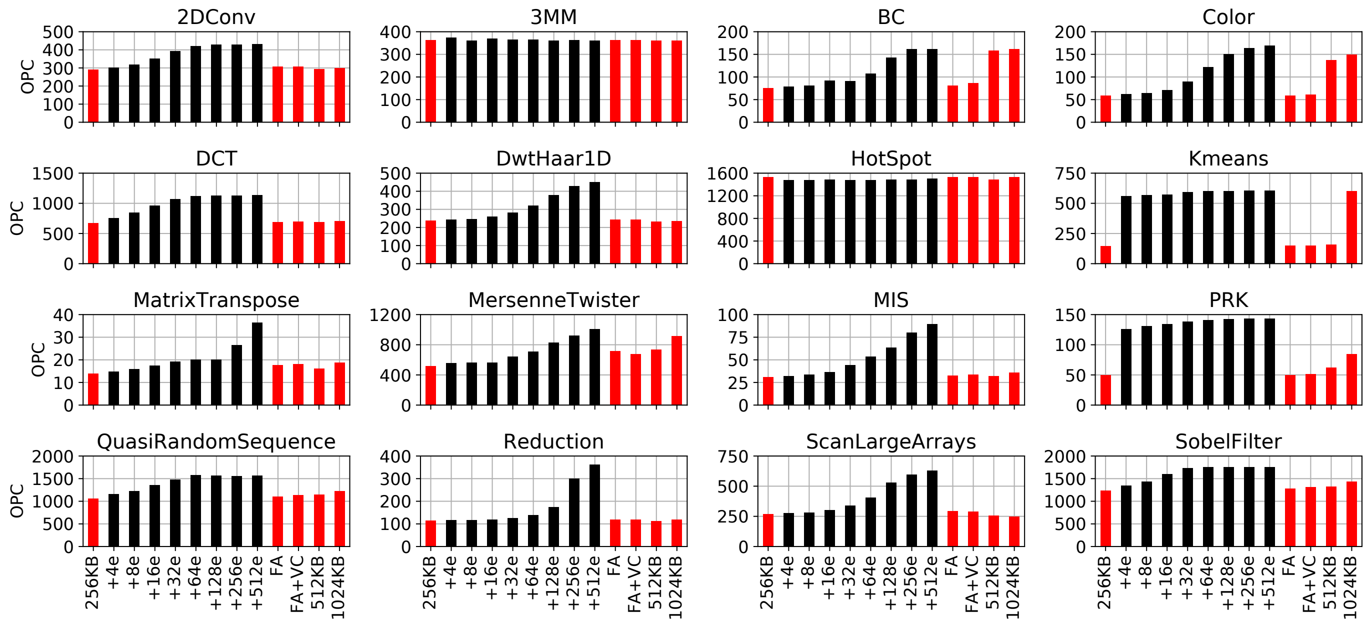

Fig. 10. Operations Per Cycle (OPC) of the RX570 across the studied applications.

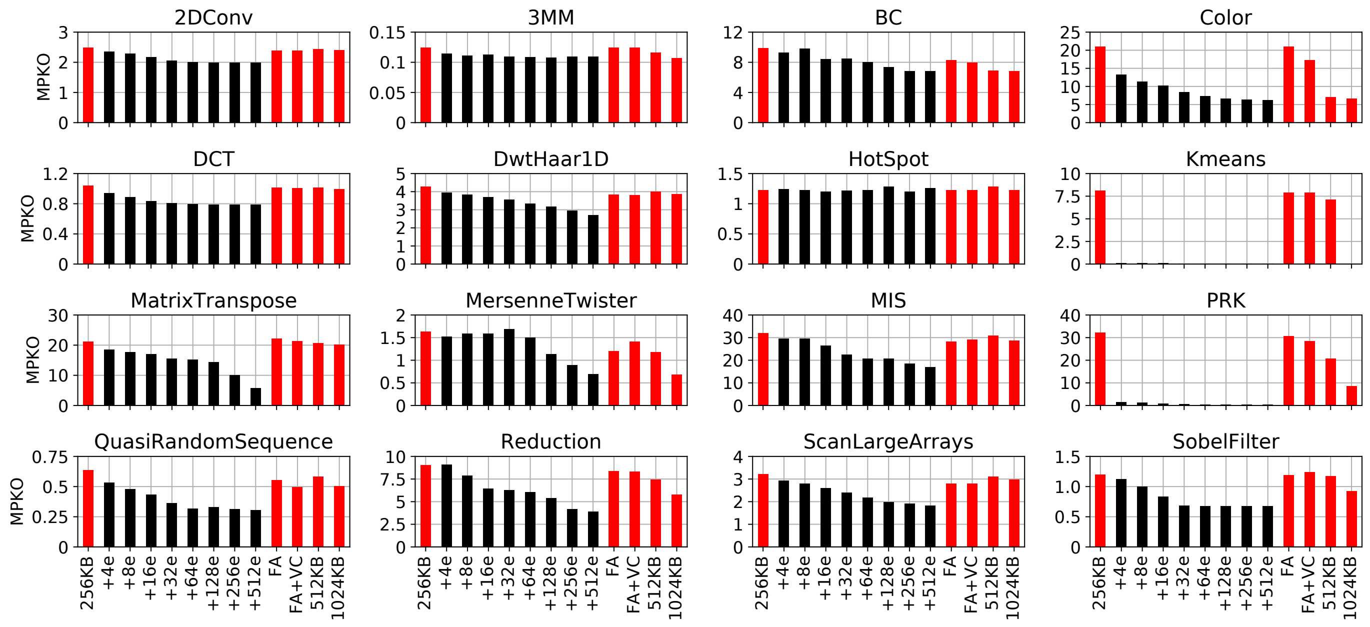

Fig. 11. Misses Per Kilo-Operation (MPKO) in the L2 cache of the RX570.

(*+512e*) on average with respect to the baseline cache. These percentages are by 22% and 50% for the 512KB and 1024KB caches, respectively. Note that all these configurations also improve the mean OPC achieved by RX540. However, as the next section will show, such a performance boost comes at the cost of a greater energy consumption and area overhead. For the FA schemes, the OPC improvement remains below 10%. To find out the reason why the RX570 GPU gets a higher improvement than the RX540, the MPKO and memory latencies for this GPU have been also studied. Figure 11 shows the MPKO results. Although the total cache capacity increases with the number of CUs, the MPKO rises in a high number (11 out of 16) of kernels with respect to the RX540 GPU because of the higher level of parallelism. Despite this fact, MPKO values of the FRC approach are similar to those of the RX540 GPU, with average MPKO reductions from 20% (*+4e*) to 48% (*+512e*) over the baseline.

Memory latencies are reduced in the RX570 GPU, as shown in Figure 12, because the memory traffic is distributed among more memory controllers. As a consequence, this GPU brings higher OPC improvements. Moreover, in the RX570, the FRC approach almost eliminates the eviction related latency and, in general, it is able to drop latency close to zero across most of the applications (remember that this latency does not include the main memory module access time). Therefore, memory contention is not an issue in the RX570, which enables further throughput improvements thanks to the MLP increase achieved by the proposal.

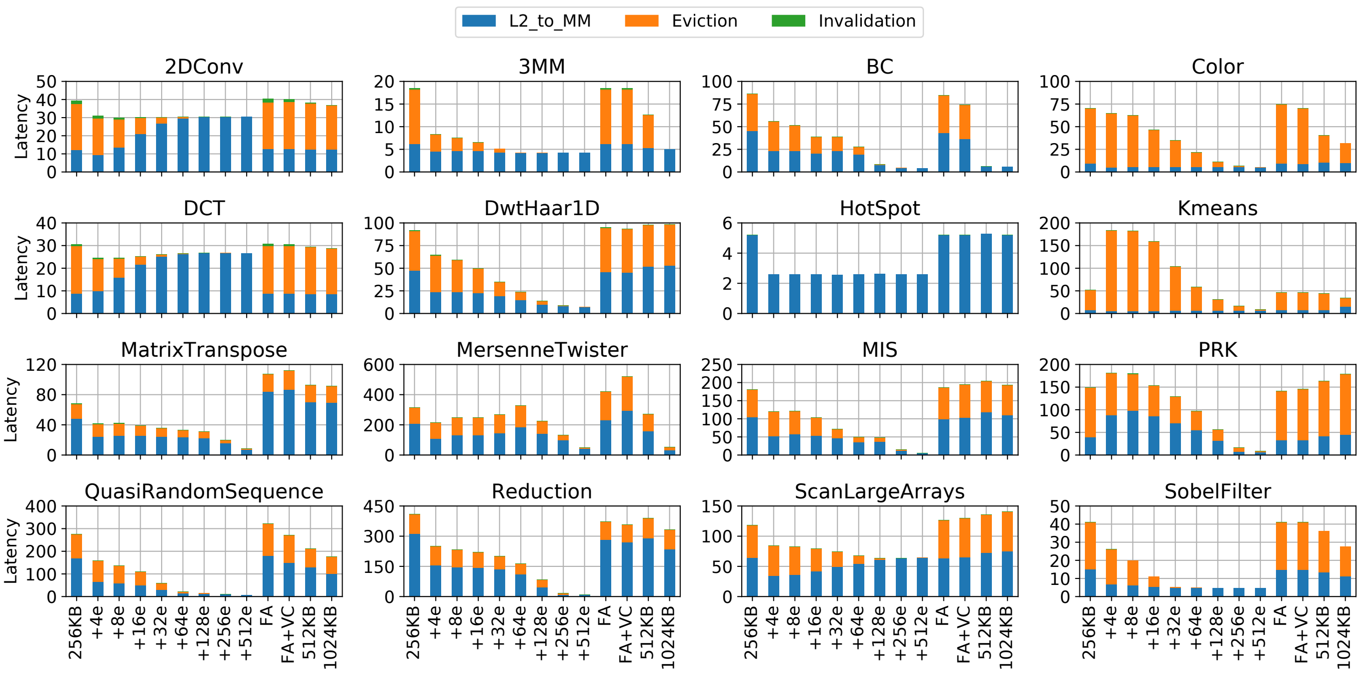

Fig. 12. Average L2 miss latency (excluding main memory access time) in processor cycles for the RX570.

5.4 Energy Consumption

This section presents the methodology used to estimate both static and dynamic energy. Then, energy results are discussed for the larger RX570 GPU. We restrict the study to such a GPU because its size helps understand the impact of our proposal in high-performance computing.

5.4.1 Methodology

The three main FRC operations are accesses, fetches, and swaps. The FRC is accessed on every L2 tag miss, which triggers an FRC tag look-up. Upon a hit, the requested block is read from the L2 data array. A fetch operation causes a write operation of the fetched block from main memory to either the FRC or the L2 cache, depending on whether the target FRC set has free entries to allocate the incoming block or not, respectively. Note that such a write operation involves writing both the tag and data arrays. Finally, a swap operation is performed after an FRC fetch, and involves four steps: i) reading the victim block from L2, ii) reading the fetched block from the FRC, iii) writing the FRC block to L2, and iv) writing the L2 block to the FRC.

CACTI has been used to quantify the dynamic energy of each type of operation. Then, these numbers are multiplied by the number of times that each operation occurs during the entire kernel execution. Static energy overheads of L2 and FRC caches are quantified considering the execution time. Execution related events have been gathered from Multi2Sim simulations.

5.4.2 Results

Figure 13 plots the total energy consumption (in mJ) of the baseline, FRC, 2× sized, and 4× sized L2 caches. The L2 and FRC energy contributions are split into static and dynamic energy. The FRC dynamic energy is in turn divided into access, fetch, and swap expenses. In addition, the dynamic energy of a 2GB GDDR main memory (MM) module is also studied.

Compared to the energy consumption of the L2 and FRC caches, the consumption of the main memory represents a significant fraction of the overall consumption in most benchmarks and cache configurations. By reducing the number of accesses to this device, the FRC approach reduces such expenses over the conventional schemes. Some benchmarks showing this behavior are `Color`, `MersenneTwister`, and `Reduction`. Moreover, this energy contribution is mostly eliminated in `Kmeans` and `PRK`.

Focusing exclusively on the L2 and FRC caches, the static expenses dominate over dynamic expenses in most applications. This is mainly due to dynamic energy is consumed only upon a cache access, whereas static energy is consumed along time regardless of the cache is being accessed or not. In addition, the accesses to the tag and data arrays of both the L2 and FRC caches are serialized, meaning that the data array is only accessed in case of tag hit, which helps mitigate dynamic energy.

As expected, static energy increases with the L2 cache size. In comparison, the much smaller and less associative FRCs present low static energy consumption. In fact, FRC configurations present much lower static energy than the baseline in some applications like `DwtHaar1D` and `MatrixTranspose`. This is because the FRC approach highly improves the system performance in such kernels (see Section 5.3); thus, the number of execution cycles and static energy are significantly reduced over the baseline. In addition, FRC configurations with 256KB L2 caches consume much less static energy per cycle than conventional schemes with 512KB and 1024KB caches.

Compared to the baseline approach, those kernels with a heavy use of FRC entries like `2DCONV` and `MersenneTwister` increase the dynamic consumption with the number of FRC entries, especially due to swaps, which translate to up to 4 different cache operations as mentioned above. Nevertheless, notice that, despite FRCs consuming more dynamic energy than the baseline, the total consumption is very similar (e.g., `2DCONV`) or even reduced in some kernels (e.g., `MersenneTwister`) thanks to energy

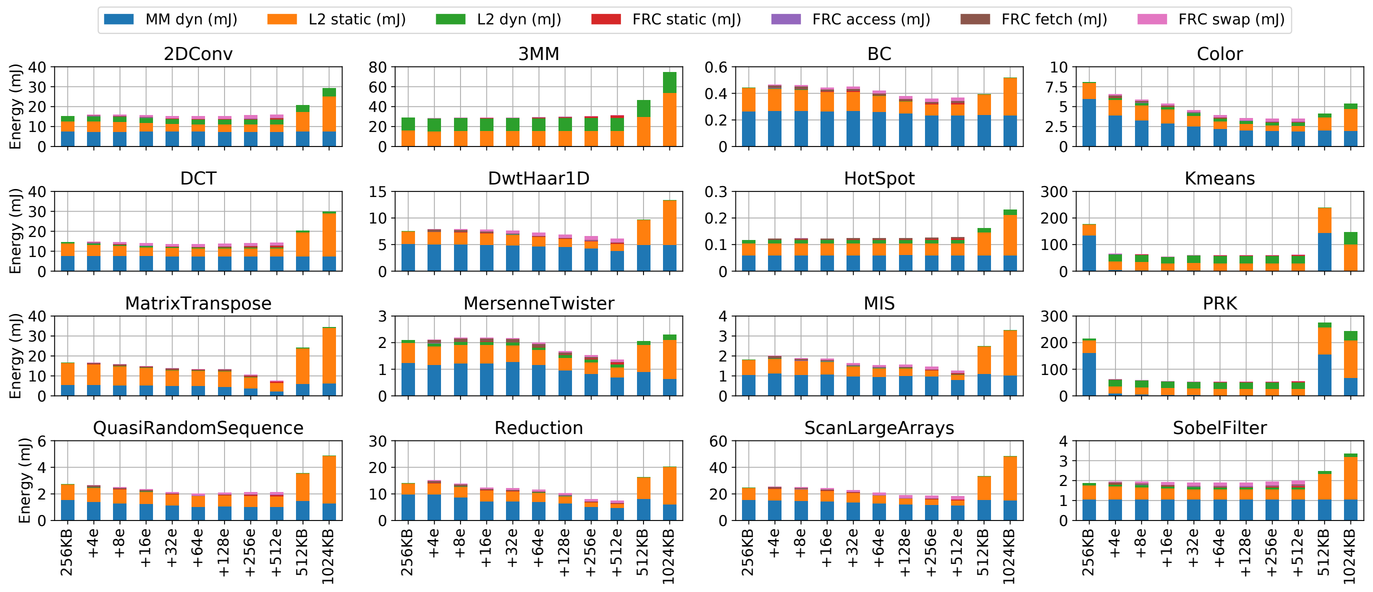

Fig. 13. Energy consumption of the RX570 including the L2 cache banks and the main memory.

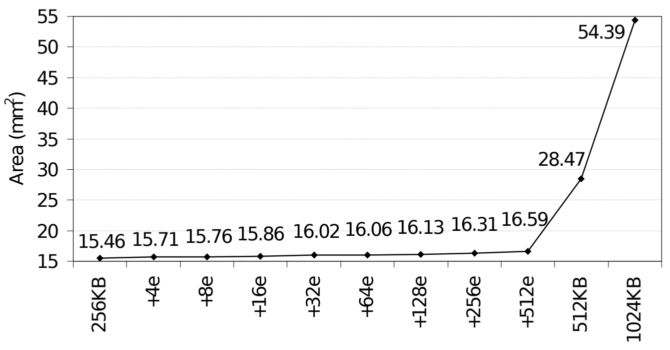

Fig. 14. Area (in mm^2) of the cache configurations for the RX570 GPU.

savings in both the dynamic main memory and static L2 energy. Furthermore, the total energy of FRC caches does not surpass that of 512KB and 1024KB caches.

Overall, the FRC approach obtains energy savings from 49% (*+4e*) to 57% (*+512e*) on average with respect to the baseline cache. Compared to the 512KB L2 cache, these percentages grow up to 62% and 67%, respectively.

5.5 Area Estimation

This section analyzes the area requirements of the proposed FRC approach. The area numbers include not only the tag and data arrays of the modeled caches, but also the cache controller peripherals (e.g., comparators, decoders, multiplexers, and sense amplifiers). Figure 14 shows the area (in mm^2) of the studied cache configurations. The presented numbers are for the RX570 GPU and refer to the 8 L2 cache banks plus the coupled FRC caches with each bank (if any).

The area overhead of the FRC schemes ranges from 1.6% (+4e) to 7.3% (+512e) compared to the baseline L2 cache without FRC. Nevertheless, these overheads are largely reduced compared to those of the much larger 512KB and 1024KB caches, whose cache capacities would translate into 4096 and 8192 FRC entries, respectively, per bank. The area overheads of these caches are up to 84.2% and 251.9%, respectively, over the baseline scheme.

5.6 Impact on Performance of Improved Memory Subsystem and Clock Frequency

In the Vega architecture [4], the most recent GPU generation from AMD to date, the GDDR5 main memory modules from the Polaris architecture are replaced with HBM modules in the GPU package. The HBM technology offers a higher memory bandwidth compared to GDDR5. In addition, the recent trend in new GPU generations is not only to improve the memory subsystem but also the GPU clock frequency.

This section evaluates the performance behavior of the FRC approach under a Vega64 GPU, which consists of 64 CUs, 16 L2 cache banks, and a clock frequency of 1.5GHz. This study not only evaluates scalability under additional computational power and memory subsystem capabilities (see Section 5.3) but also an improved memory subsystem and a higher clock frequency has been considered.

Figure 15 shows the OPC results for the Vega64 GPU. The Vega64 presents better OPC values with respect to the RX540 and RX570 across all the studied benchmarks thanks to the improved computational and memory capabilities and higher memory bandwidth. This fact does not prevent the FRC from boosting the OPC over the baseline cache in most applications. Although the average OPC improvements are not as high as those from the RX540 and RX570 GPUs, the FRC still boosts the OPC from 16% (*+4e*) to 54% (*+512e*) compared to the 256KB L2 cache. In this study, the 4× sized cache also reaches an average OPC improvement of 54% over the baseline. However, such a performance would be achieved with a greater energy consumption and area as discussed above.

6 Related Work

Prior work focusing on the GPU memory subsystem can be classified into works aimed at primarily improving either system performance or energy consumption, which are summarized in this section.

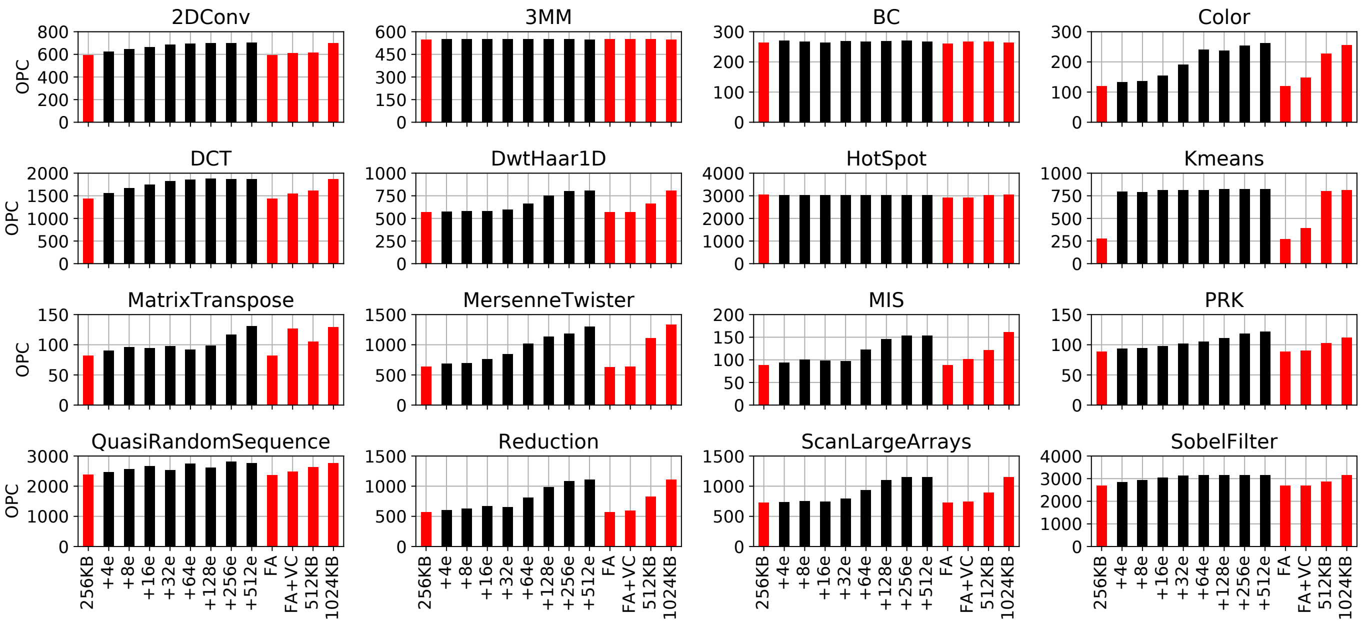

Fig. 15. Operations Per Cycle (OPC) of the Vega64 across the studied applications.

6.1 System Performance

The GPU memory subsystem performance has been widely analyzed in recent years from different perspectives including cache bypassing techniques [18], [20], [28], and optimization techniques of the memory subsystem design [17], [10], [24], [39], [13]. This section summarizes prior work in this regard.

Elastic-Cache [17] supports fine-grained L1 cache line management for 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 by many kernels. Gebhart et al. [10] 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 [24] 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. Taylor and Chang [22] investigate the effectiveness of adding a victim buffer to the L1 cache, and show that victim buffers with a relatively low number of lines obtain the same performance as doubling the L1 cache size. Wang et al. [39] 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. MRPB [13] is a memory-request priorization buffer that allows reordering and bypassing memory requests before they access the L1 cache. After being captured by the MRPB buffer, memory requests are released into the cache in a cache-friendly order to reduce cache thrashing and stalls.

Other research work has focused on memory and wavefront scheduling strategies [23], [34], [31].

6.2 Energy Consumption

Research addressing energy consumption in on-chip GPU caches has been done from different points of view including adaptive cache management techniques such as bypassing, thread throttling, indexing schemes, fine-grained fetching, and power-gating techniques [36], [9], [16], [30], [40], using alternative memory technologies to SRAM for on-chip storage [15], [32], and the proposal of additional on-chip memory structures [33].

Tian et al. [36] prevent streaming one-time-use blocks into the L1 cache with a dynamic bypass prediction technique. The proposed technique saves energy by avoiding useless cache insertions and evictions. Chen et al. [9] propose to protect the memory hierarchy from contention with a bypass policy based on reuse distance. Besides, this policy is combined with a thread throttling technique that dynamically controls the active number of threads in order to mitigate the contention and resource congestion. Reducing both the memory hierarchy contention and congestion translates into energy savings with respect to a conventional approach. IACM [16] is an integrated architecture combining Chen's bypassing and thread throttling techniques with an L1 cache indexing scheme. IACM dynamically determines the cache indexing bits that can mitigate cache thrashing and contention based on the runtime information of GPU kernels. LAMAR [30] is a technique that facilitates a fine-grained control of DRAM data fetches for those blocks with low spatial and temporal locality, reducing the energy-hungry traffic between on- and off-chip memory. This technique is combined with a bloom-filter predictor to adjust the fetching granularity at runtime. Wang et al. [40] mitigate the leakage energy consumption by putting both L1 and L2 caches in a state-retentive sleep mode when there are no ready threads to be scheduled and no memory requests, respectively. The effectiveness of the mechanism lies in the fact that the power on/off latencies are completely hidden.

Alternative high-density and low-leakage memory technologies have been used to implement energy-efficient GPU memory subsystems. Jing et al. [15] implement the GPU register file, shared memory, and L1 cache with eDRAM technology. The refresh penalty introduced by eDRAM is mitigated with the proposal of refresh mechanisms assisted by the compiler. Samavatian et al. [32] use STT-RAM technology to implement L2 caches. The main shortcomings of STT-RAM are the high energy and latency of write operations, which are addressed by reducing the data retention time thanks to the kernel data behavior.

Finally, additional memory structures have been also used for energy efficiency. In [33], 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 and save on-chip energy. By leveraging intrinsic characteristics of GPU programming models, these caches are kept non-coherent to avoid incurring additional overheads.

7 Conclusions

This paper has shown that the way Last-Level Cache (LLC) misses are handled in typical GPUs acts as a major performance limiter. To deal with this shortcoming, this work has presented a novel GPU cache subsystem design that leverages a tiny Fetch and Replacement Cache-like structure (FRC) between the 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 MLP, improving the lifetime of the victimized blocks and removing eviction latencies from the critical path. Moreover, the small size of the FRC provides additional benefits regarding energy consumption and area compared to merely enlarging the LLC size.

The FRC attacks by design three main cache performance related issues, which results in a much better LLC cache management: i) it reduces the number of 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 cache miss rises, and iii) it increases the MLP by unclogging new block requests whose target line is already being replaced.

Experimental results have shown that, compared to a conventional LLC design, the FRC increases the average OPC by 67%. In addition, the proposal also presents a high scalability, since it provides more performance benefits in a larger GPU, whose average OPC grows up to 118% over the baseline. Moreover, compared to a GPU using the recent HBM technology to implement the main memory modules, FRC improves the average OPC up to 54% over the conventional design. Such benefits come from a reduction of MPKO due to a higher availability of the contents of victim blocks as well as a reduction of miss latencies due to removing unnecessary serializations and eviction penalties from the critical path. We also found that in some kernels, latency increases because of the higher MLP, which causes additional contention accessing main memory. Nevertheless, this latency increase is not enough to constrain the performance improvements given by the MLP growth.

Results have also shown that the energy overhead of adding a small FRC with just tens of entries is largely compensated by its effectiveness, reaching energy savings up to 57% compared to the conventional design. These savings come with less than a 7.3% of LLC area increase.

Finally, evaluating the FRC approach considering also private L1 caches is planned as for future work.

Acknowledgments

This work has been supported by the Spanish *Ministerio de Ciencia, Innovación y Universidades* and the European ERDF under Grants T-PARCCA (RTI2018-098156-B-C51), and TIN2016-76635-C2-1-R (AEI/ERDF, EU), by the *Universitat Politècnica de València* under Grant SP20190169, and by the gaZ: T58_17R research group (Aragon Gov. and European ESF).

References

[1] AMD. AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) . http://developer.amd.com/sdks/amdappsdk/, 2011.
[2] AMD. AMD Graphics Cores Next (GCN) Architecture White Paper. https://www.amd.com/documents/gcn_architecture_whitepaper.pdf, 2012.
[3] AMD. Radeon^{TM}. Dissecting the Polaris Architecture. http://radeon.com/_downloads/polaris-whitepaper-4.8.16.pdf, 2016.
[4] AMD. Radeon's next-generation Vega architecture. https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf, 2017.
[5] F. Candel, S. Petit, J. Sahuquillo, and J. Duato. Accurately Modeling the On-chip and Off-chip GPU Memory Subsystem. *Elsevier Future Generation Computer Systems*, 82:510–519, 2018.
[6] F. Candel, S. Petit, A. Valero, and J. Sahuquillo. Improving the GPU Cache Hierarchy Performance with a Fetch and Replacement Cache. In *Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, pages 235–248, 2018.
[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 185–195, 2013.
[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.
[9] X. Chen, L. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355, 2014.
[10] 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 *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 96–106, 2012.
[11] A. Glenis and S. Petridis. 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*, pages 181–186, 2014.
[12] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.
[13] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture*, pages 272–283, 2014.
[14] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR*, 2018.
[15] N. Jing, L. Jiang, T. Zhang, C. Li, F. Fan, and X. Liang. Energy-Efficient eDRAM-Based On-Chip Storage Architecture for GPGPUs. *IEEE Transactions on Computers*, 65(1):122–135, 2016.
[16] K. Y. Kim, J. Park, and W. Baek. IACM: Integrated Adaptive Cache Management for High-Performance and Energy-Efficient GPGPU Computing. In *Proceedings of the IEEE 34th International Conference on Computer Design*, pages 380–383, 2016.
[17] B. Li, J. Sun, M. Annavaram, and N. S. Kim. 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*, pages 82–91, 2017.
[18] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th International ACM Conference on Supercomputing*, pages 67–77, 2015.

[19] Y. Liang, X. Xie, G. Sun, and D. Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1677–1690, 2015.

[20] Y. Liang, X. Xie, Y. Wang, G. Sun, and T. Wang. Optimizing Cache Bypassing and Warp Scheduling for GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1560–1573, 2018.

[21] X. Mei and X. Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[22] E. M.Taylor and D. W.Chang. Studying Victim Caches in GPUs. In *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 394–398, 2018.

[23] 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*, 22(8):1803–1814, 2014.

[24] S. Mu, Y. Deng, Y. Chen, H. Li, J. Pan, W. Zhang, and Z. Wang. IBOM: An Integrated and Balanced On-Chip Memory for High Performance GPGPUs. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):586–599, 2018.

[25] C. Nugteren, G. van den Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *IEEE 20th International Symposium on High Performance Computer Architecture*, pages 37–48, 2014.

[26] NVIDIA. NVIDIA GeForce GTX 980 Whitepaper. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014.

[27] NVIDIA. NVIDIA Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[28] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram. APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 191–203, 2016.

[29] L.-N. Pouchet. Polybench: The Polyhedral Benchmark Suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[30] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 86–98, 2013.

[31] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83, 2012.

[32] M. H. Samavatian, M. Arjomand, R. Bashizade, and H. Sarbazi-Azad. Architecting the Last-Level Cache for GPUs Using STT-RAM Technology. *ACM Transactions on Design Automation of Electronic Systems*, 20(4):55:1–55:24, 2015.

[33] A. Sankaranarayanan, E. K. Ardestani, J. L. Briz, and J. Renau. An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 9–14, 2013.

[34] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, pages 174–185, 2015.

[35] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 51–62, 2008.

[36] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez. Adaptive GPU Cache Bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, pages 25–35, 2015.

[37] Top500.org. Top500 Supercomputer Sites, http://top500.org.

[38] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 335–344, 2012.

[39] J. Wang, F. Fan, L. Jiang, X. Liang, and N. Jing. Incorporating Selective Victim Cache into GPGPU for High-performance Computing. *Wiley Concurrency and Computation: Practice and Experience*, 29(24):1–11, 2017.

[40] Y. Wang, S. Roy, and N. Ranganathan. Run-Time Power-Gating in Caches of GPUs for Leakage Energy Savings. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, pages 300–303, 2012.

Francisco Candel received the BS and MS degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2012 and 2014, respectively. He is currently working towards a PhD degree at the Department of Computer Engineering (DISCA) of the same university. His PhD research focuses on GPU modeling and efficient memory hierarchies for future GPUs.

Alejandro Valero received the Ph.D. degree in Computer Engineering from the Universitat Politècnica de València, Spain, in 2013. From 2013 to 2015, he was a Visiting Researcher with Northeastern University, Boston, Massachusetts, and the University of Cambridge, United Kingdom. Since 2016, he has been an Assistant Professor with the Department of Computer Science and Systems Engineering at the University of Zaragoza, Spain. His current research interests include GPU architecture, memory hierarchy design, energy efficiency, and reliability. In 2012, Dr. Valero received the Intel Doctoral Student Honor Program Award.

Salvador Petit (M'07) received the PhD degree in computer engineering for the Universitat Politècnica de València (UPV), Spain. Since 2009, he has been an Associate Professor with the Computer Engineering Department, UPV, where he has been teaching several courses on computer organization. He has authored over 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, GPU architecture, and resource management. Dr. Petit is a member of the IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.

Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the Universitat Politècnica de València, Spain, all in computer engineering. He is a Full Professor with the Department of Computer Engineering at the Universitat Politècnica de València. He has taught several courses on computer organization and architecture. He has authored over 150 refereed conference and journal papers. His current research interests include multi- and manycore processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. Dr. Sahuquillo is a member of the IEEE Computer Society.