J Supercomput (2018) 74:2314–2328
https://doi.org/10.1007/s11227-018-2260-6

Stream data prefetcher for the GPU memory interface

Nuno Neves[1] · **Pedro Tomás**[1] · **Nuno Roma**[1]

Published online: 27 January 2018

Abstract Data caches are often unable to efficiently cope with the massive and simultaneous requests imposed by the SIMT execution model of modern GPUs. While software-aided cache management techniques and scheduling approaches were early considered, efficient prefetching schemes are regarded as the most viable solution to improve the efficiency of the GPU memory subsystem. Accordingly, a new GPU prefetching mechanism is proposed, by extending the stream computing model beyond the actual GPU processing core, thus broadening it toward the memory interface. The proposed prefetcher takes advantage of the available cache management resources and combines a low-profile architecture with a dedicated pattern descriptor specification, which is used to explicitly encode each kernel memory access pattern. The obtained results show that the proposed mechanism increases the $L1$ data cache hit rate by an average of 61%, resulting in performance speedups as high as 9.2× and consequent energy efficiency improvements as high as 11×.

Keywords GPU Prefetching · Stream-based Prefetching · Data-pattern encoding · Assisted memory access

✉ Nuno Neves
nuno.neves@inesc-id.pt

Pedro Tomás
pedro.tomas@inesc-id.pt

Nuno Roma
nuno.roma@inesc-id.pt

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol, 9, 1000-029 Lisbon, Portugal

1 Introduction

The increasing demand for computation that was observed in the last decade, allied with the critical energy consumption and cost constraints, has pushed the adoption of Graphics Processing Units (GPUs) as massively parallel and high-performance computing devices. However, despite the rather consolidated architecture and execution model that has been adopted for GPUs, the large amount of simultaneously executing threads (up to thousands) often imposes a significant pressure into their communicating infrastructures, leading to highly penalizing contentions in the memory accesses [3].

In an effort to tackle such issues, GPUs have been introducing larger and more elaborate cache hierarchies. This, in turn, allowed the deployment of a wider range of applications, but at a cost of relying on intensive, complex and hardly manageable memory access patterns [9]. On the other hand, due to the large amount of concurrently executing threads, GPUs cannot directly utilize Central Processing Unit (CPU)-like cache structures. This frequently leads to an increased contention to move data from the main memory to caches, often degrading (instead of improving) the resulting performance [8,21,23]. Moreover, given the number of requests for distinct memory regions, data locality is often poorly exploited, resulting in very low cache hit rates and increased access latencies [8,21,23].

Some emerging solutions consider the adaptation and deployment of widely known CPU-like prefetching mechanisms in GPUs, while keeping in mind that conventional CPU techniques are not particularly suited for GPUs, due to the greater amount of available parallelism (as a matter of fact, they can even degrade the performance [2,12]). Most of these solutions are deployed both at software and hardware levels and rely on feedback-driven mechanisms to adapt the prefetching scheme and its intensity to the application being executed. Some example solutions include inter-warp prefetching [12], also allied with adapted warp scheduling mechanisms [10] and adaptive intra-warp collaboration [19].

However, even though these adaptive prefetching schemes are able to solve some of the above mentioned issues, they still rely on mechanisms based on runtime access stride detection and smart prediction systems. These, in turn, require complex control schemes and a considerable amount of additional computation and training time. As a result, many of these approaches may ultimately lead to performance degradations in adverse scenarios (increased contention in the communication infrastructures when the prediction fails) and to higher energy consumption (unnecessary data requests from the main memory).

In contrast, it is herein considered the possibility of gathering and describing the memory access patterns beforehand (with compile-time analysis or through manual input) in order to deploy far simpler and more efficient data fetch mechanisms at runtime. In particular, it is proposed a stream-based prefetching mechanism that adopts explicit memory access pattern description encodings (based on [15]) and relies on a low-profile buffered prefetcher that is aggregated in the GPU memory interface. The proposed mechanism takes advantage of the Miss Status Hold Registers (MSHRs) of the $L1$ data cache of each Streaming Multiprocessor (SM), by independently prefetching the required data for a given issued cooperative thread array (CTA). It also transparently intercepts the cache miss memory requests, serving them either

with buffered prefetched data or merging them with outstanding prefetch requests in the MSHRs. Such an approach allows the deployment of an autonomous prefetching scheme that does not rely on complex monitoring and feedback-based control substructures, in turn eliminating detection overheads and reducing the amount of contention and the pressure to the memory subsystem. Hence, the main contributions of this manuscript are as follows:

- A new offline data-pattern description specification for GPU applications that is able to encode the exact memory access pattern of a given application kernel;
- A novel stream prefetching mechanism for GPU accelerators, to be integrated in their memory subsystem, to transparently perform data fetch operations, by taking advantage of the available data management resources;
- A low-profile buffered prefetcher architecture for GPU $L1$ data caches, supported by a memory address generation unit that deploys the proposed data-pattern description specification.

The proposed prefetching mechanism was implemented and thoroughly simulated with the GPGPU-Sim simulator [3]. The resulting implementation was then experimentally evaluated by considering a significant subset of the Rodinia [4] and Polybench [6] benchmark suites. The obtained results show that the proposed mechanism provides an increase of the $L1$ cache hit rate of 61% (on average), resulting in average performance speedups of $3\times$ when compared with traditional approaches. The attained performance gains allow global energy consumption savings of 39% (on average) and consequent energy efficiency improvements of $3.8\times$.

The remaining of this manuscript is organized as follows. In Sect. 2, it is presented the current state-of-the-art and a overview on GPU prefetching. Section 3 describes the proposed stream prefetching mechanism. In Sect. 4, it is detailed the stream prefetcher architecture. Section 5 presents the obtained experimental results, followed by the conclusions (in Sect. 6) addressing the main contributions and achievements.

2 Background and related work

Despite their massively parallel computing structure, GPUs can only partly mitigate the inherent pressure in the main memory subsystem by exploiting a Single-Instruction Multiple-Thread (SIMT) execution paradigm, where a massive amount of threads is launched at each SM, in groups denoted as *thread blocks* (or *CTAs*). The threads of each CTA are then executed in fixed-sized batches (named *warps*), in which all threads simultaneously execute the same instruction. In order to deal with the long memory access (and inherent execution) latencies, warps are switched and enqueued while data dependencies and outstanding memory requests are resolved. This is essentially accomplished by increasing the amount of on-the-fly parallelism, in order to hide the instruction execution latency behind the computation of other threads.

However, due to the long access times of the global memory, the SIMT architecture and the warp switching mechanisms, by themselves, can hardly mitigate the imposed overheads, especially when dealing with general purpose applications with complex access patterns. In the particular case of NVIDIA GPU architectures [16,17], each SM typically integrates $L1$ CTA-private caches (constant, texture and data) and a

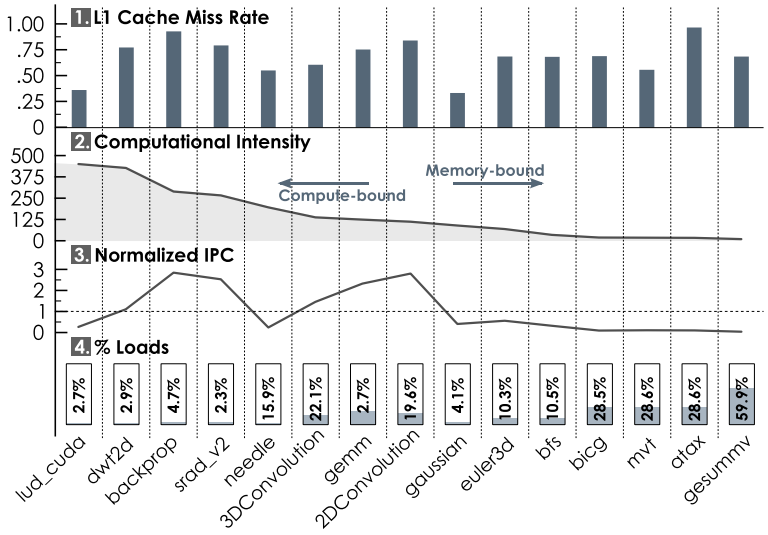


Fig. 1 Application profiles [computational intensity (issued instructions/memory accesses), cache miss rate, percentage of issued load instructions and normalized IPC] for subsets of the Rodinia [4] and Polybench [6] benchmark suites

local shared memory used for inter-CTA communication. Due to the high volume of simultaneous requests, cache misses are registered in dedicated Miss Status Hold Registers (MSHRs) and scheduled to the subsequent memory level [16]. Furthermore, in order to efficiently manage the available memory bandwidth, requests to contiguous memory positions are grouped together in coalescing units in the $L1$ caches [16].

2.1 GPU cache performance

The addition of cache memories to the base SM architecture allowed for a significant mitigation of the contention and long memory access latencies. However, they cannot efficiently deal with the massive amount of threads executed by each SM and they are not well suited to deal with the complexity and demand of the memory access patterns of certain high-performance computing (HPC) kernels. This problem is clearly highlighted by the observed $L1$ cache miss rates (up to 94%), and by its impact on the system's performance (shown in Fig. 1 for representative subsets of the Rodinia [4] and Polybench [6] benchmark suites). In particular, besides thread management, scheduling and profiling techniques [13,20], only emergent prefetching techniques [2,10,12,19] have proven to be able to deal with such drawbacks.

2.2 GPU prefetching overview

This trend on the adoption of new prefetching techniques is justified by the inherent characteristics of current HPC applications. In fact, the straightforward introduction of prefetching mechanisms to execute memory-bound applications (with lower computational intensity—see Fig. 1) inherently allows remarkable improvements of the

performance and cache efficiency [15]. However, cache inefficiencies resulting from complex memory access patterns (namely poor spatial and temporal locality) can only be significantly mitigated with aggressive predictive techniques and software-aided control, monitoring and feedback mechanisms to manage the prefetching procedure [2,10,12].

An early solution proposed a many-thread aware prefetching mechanism [12] that combines both software and hardware approaches to hide the communication latency. To achieve this, inter-thread prefetching is performed by making a group of threads to prefetch the data for subsequently scheduled threads. This scheme is deployed at both software and hardware levels, together with an adaptive training algorithm for automatic management. In [10], the authors acknowledge that existing warp scheduling policies cannot cope with conventional data prefetching, since the scheduling of consecutive warps easily incurs prefetch requests for common memory regions, leading to increased communication and contention. In order to tackle this issue, the authors propose an alternative warp scheduling policy, allied with a simple prefetcher, to separate, in runtime, the scheduling of consecutive warps. Hence, by organizing distant warps in different fetch groups, they do not immediately execute one after the other, therefore avoiding overlapping prefetch requests.

To mitigate possible early evictions of prefetched data, [11] proposes a prefetching mechanism, targeting graph algorithms, that detects specific load instructions in hardware and injects instructions into the pipeline to prefetch data into spare (unused) registers. The prefetcher is combined with a compile-time mechanism to identify target loads.

Although successful, the previously referred software-based approaches still rely on complex hardware control structures and inter-thread prefetching operations, which can be ineffective in certain applications [2]. To address such concerns, a low-overhead prefetcher is proposed in [19], which dynamically adapts to the address patterns found in both graphics and scientific applications. This prefetcher deploys a dynamic and adaptive prediction scheme that adjusts the prefetching distance through an intra-warp collaboration approach.

However, due to their predictive nature and the required control overheads, not only do these techniques reportedly show some non-neglectable prefetching inaccuracies [19], but they also fall short when dealing with complex memory access patterns and they impose an increased amount of hardware resources (depicted in Fig. 2, 1). In contrast, the stream-like processing paradigm usually adopted at the GPU SM processing structures can be extended toward the memory interface [18], with the allied advantage of not relying on complex predictive structures and elaborate feedback-based monitoring techniques. However, such a streaming paradigm must be self-managed and transparent to the GPU SMs architecture, as well as easily integrated into the memory subsystem (as shown in Fig. 2, 2). Furthermore, in order to avoid prediction overheads and prefetching inaccuracies, the prefetcher should be implemented by relying on explicit access patterns for the kernel in execution. Such a stream-like paradigm can be supported either on programmer-provided pragmas or by specific profiling and pre-analysis of the code, to extract/model and encode the memory access patterns of each kernel, subsequently feeding them to the on-chip prefetchers [15].

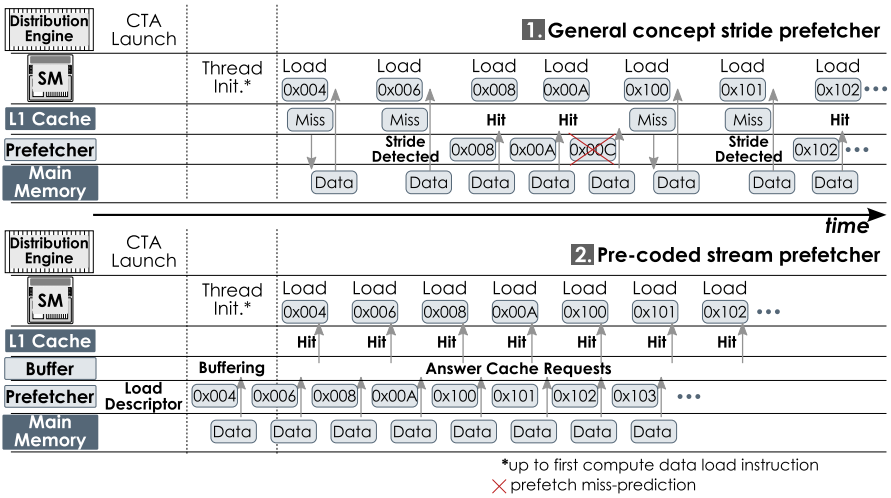


Fig. 2 Conceptual operation of a typical predictive prefetcher against a pattern-encoded approach

3 Proposed GPU Stream-based prefetching

The proposed mechanism is based on application-specific data-pattern descriptions, explicitly defined by the programmer (or extracted by the compilation toolchain). In accordance, the adopted prefetching structures are programmed with the memory access pattern of a given kernel, hence eliminating the need for predictive runtime analysis and avoiding the deployment of complex prefetch decision control structures.

Several approaches can be considered regarding the granularity of a prefetching mechanism and the context level at which it is applied. In fact, it is common to exploit prefetching techniques at a thread-level granularity [12], which in practice incurs a warp-level prefetching, since all threads of a given warp simultaneously perform the memory accesses. However, although such a fine-grained prefetching allows data to be fetched in the exact same order in which warps are scheduled, it also requires active complex control and monitoring infrastructures (including modifications to the scheduler itself [2,10]).

On the other hand, if a more coarse-grained approach is considered, other transparent and fully autonomous prefetching mechanisms can be explored. In particular, a CTA-level prefetching granularity is adopted by the proposed mechanism, targeting the deployment of a low-profile and transparent stream-based communication paradigm. Hence, all memory access pattern descriptions are encoded with a CTA-level granularity, with the valid assumption that their corresponding warps are initially scheduled in order and switched in a round-robin fashion [3,21]. By designing the proposed mechanism in such a way that it fetches and buffers data as soon as a CTA is launched, any warp-reordering effects are mitigated by making data available as soon as possible.

3.1 Data stream pattern representation

Usually, the memory access patterns of most general purpose GPU applications are explicitly coded by the programmer, since the development and optimization of the kernels requires specific code and data transformations in order to attain efficient per-thread and per-warp memory access sequences, therefore maximizing coalescing and minimizing contention. In a similar approach, the memory access patterns can be extracted and manually encoded in a pattern description code. Notwithstanding, although not herein considered, such procedures can also be performed with the aid of compile-time analysis algorithms, including memory access tracing [1,9], complexity analysis [22] and polyhedral analysis [7]. Similar techniques will be considered in future implementations, regarding the integration of these tools in compile-time toolchains of parallel programming languages (such as OpenCL and OpenACC).

Notwithstanding, the adoption of an explicit data-pattern representation allows indexing an ample subset of regular memory access patterns, since most complex patterns can be exactly described by an aggregate of n-dimensional affine functions [5]. Accordingly, each data stream can be defined by a set of n-dimensional *descriptors*, each encapsulating the set of parameters required to generate the sequence of addresses. This way, any data-pattern descriptors are embedded and sent together with the binary of the compiled kernel code, either: (i) by encoding all patterns/kernel phases in a single hierarchical descriptor, merging all data patterns in the same encoding; or (ii) by providing a list of descriptors to the SM (ordered by execution phase) that are resolved in sequence. Then, when a given CTA is launched to a SM, the descriptor code is extracted and sent to a prefetcher module (see Sect. 4), along with the parameters identifying the CTA. Subsequently, the prefetcher decodes and initializes its descriptors and initiates the corresponding data fetch procedure.

Accordingly, a CTA-level prefetching encoding specification is herein proposed (based on [15]). It adopts a three-dimensional (n=3) base descriptor, in order to match as much as possible with the thread distribution topology in a CUDA block [16], hence ensuring an efficient description of most regular patterns of general purpose applications. As such, any given memory access pattern can be represented by the tuple presented in Eq. 1, specifying the starting address of the first memory block (OFFSET), the size of each contiguous block (HSIZE), the starting position of the next contiguous block with relation to the previous (STRIDE), the number of repetitions of the two previous parameters (VSIZE), the starting of the next pattern represented by the previous parameters (SPAN) and the number of repetitions of the four previous parameters (DSIZE).

$$\textbf{Descriptor Specification} : \{\mathit{offset}, \mathit{hsize}, \mathit{stride}, \mathit{vsize}, \mathit{span}, \mathit{dsize}\} \quad (1)$$

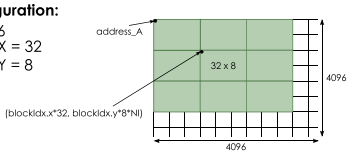
The affine nature of the proposed encoding is able to exactly describe deterministic memory access patterns. Furthermore, it can also be used to accelerate irregular applications that do not present deterministic memory accesses before runtime. This is done by encoding a slightly larger memory region where data are likely to be accessed at runtime, instead of the exact access sequence. Furthermore, such an approach takes advantage of the data organization optimizations that are typically performed in the

2D Convolution Kernel:

```
__global__ void Convolution2D_kernel(DATA_TYPE *A, DATA_TYPE *B)
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    DATA_TYPE c11, c12, c13, c21, c22, c23, c31, c32, c33;

    c11 = +0.2;  c21 = +0.5;  c31 = -0.8;
    c12 = -0.3;  c22 = +0.6;  c32 = -0.9;
    c13 = +0.4;  c23 = +0.7;  c33 = +0.1;

    if ((i < NI-1) && (j < NJ-1) && (i > 0) && (j > 0))
    {
        B[i * NJ + j] =
          c11 * A[(i - 1) * NJ + (j - 1)] + c21 * A[(i - 1) * NJ + (j + 0)]
        + c31 * A[(i - 1) * NJ + (j + 1)] + c12 * A[(i + 0) * NJ + (j - 1)]
        + c22 * A[(i + 0) * NJ + (j + 0)] + c32 * A[(i + 0) * NJ + (j + 1)]
        + c13 * A[(i + 1) * NJ + (j - 1)] + c23 * A[(i + 1) * NJ + (j + 0)]
        + c33 * A[(i + 1) * NJ + (j + 1)];
    }
}
```

Kernel Configuration:

NI = NJ = 4096
BLOCK_DIM_X = 32
BLOCK_DIM_Y = 8



CTA Generic Data Pattern Descriptor[1]:
{address_A + blockIdx.x * blockDim.x + blockIdx.y * blockDim.y * NI, blockDim.x, NJ, blockDim.y, 0, 1}

Descriptor Encoding (w/ constants):
{address_A + blockIdx.x * 32 + blockIdx.y * 8 * 4096, 32, 4096, 8, 0, 1}

Decoded Descriptor at CTA launch:
Example CTA id = (1,1): {address_A + 32 + 8 * 4096, 32, 4096, 8, 0, 1}

[1] {offset, hsize, stride, vsize, span, dsize}

Fig. 3 Example data-pattern descriptor specification and its decoding for the 2D Convolution application, of the Polybench [6] benchmark suite

development of irregular applications, allowing an initial population of the memory resources with the targeted memory region.

3.2 Pattern description code integration

According to the GPU's SIMT execution model, the data processed by each thread in a CTA can be indexed by using predefined system variables containing the corresponding thread and CTA identifiers. Hence, although the assigned kernel code is the same, each CTA initializes the values of such variables with its own runtime identification parameters, thus defining the memory regions that each thread will access. By following the same principle, the fields of the adopted descriptor specification can be initially encoded by taking advantage of the same CTA identification parameters, since the memory access pattern of a given kernel is essentially the same for all of its CTAs.

Figure 3 presents an example corresponding to the 2D Convolution benchmark, from the Polybench [6] suite. In order to perform the required computations, an input matrix is divided in blocks, which are assigned to different CTAs. There, each thread calculates a single element of the output matrix. The indexing of each element is performed (as usual) with the native block identification and dimension primitives of the CUDA programming language (as depicted in Fig. 3). As such, it is possible to also use those variables to encode a generic descriptor representing the data pattern of each CTA and, in turn, decode it for a specific CTA when it is launched to the GPU (see Fig. 3).

4 Data stream prefetcher

The proposed stream prefetching mechanism (defined in Sect. 3) was implemented and integrated in the adopted NVIDIA Fermi^{TM} GPU architecture [16], in parallel with the existing $L1$ data caches (see Fig. 4). Nevertheless, although this architecture was selected due to its support by GPGPU-Sim [3] simulator, the proposed mechanism is

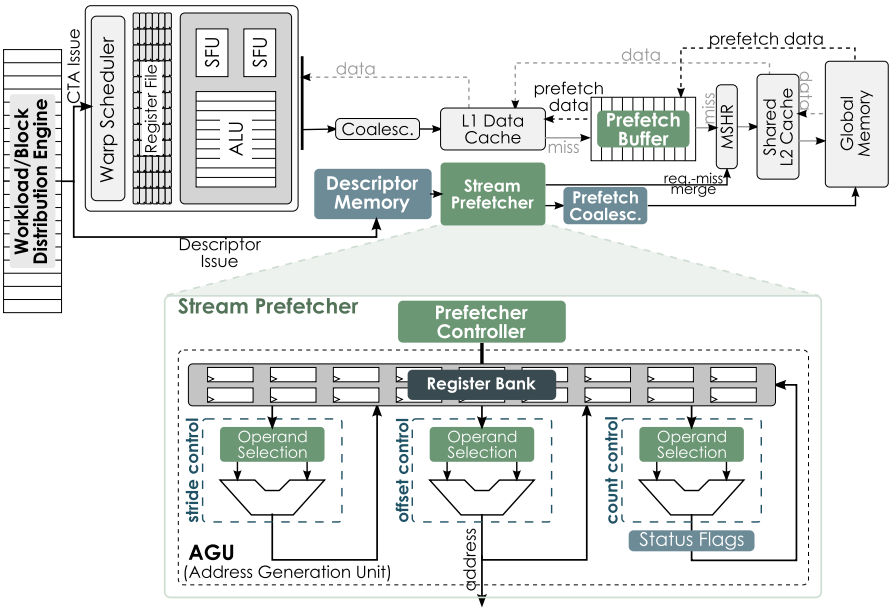


Fig. 4 Integration of the proposed stream prefetching mechanism in the SM's *L*1 memory sub-hierarchy and prefetecher architecture

architecture-independent and can be straightforwardly implemented in recent NVIDIA architectures [17]. In fact, neither the *L*1 data/texture cache unification nor the new dynamic parallelism features, recently introduced in NVIDIA GPUs [17], affect or compromise the proposed prefetcher's integration in the memory hierarchy.

The implemented stream prefetcher is responsible for issuing prefetch requests, encoded with the proposed data-pattern descriptors. It also intercepts and merges cache miss requests with prefetch requests and serves them with prefetched data. Its architecture comprehends: (i) a dedicated controller module, responsible for accepting incoming descriptors and for initializing them with CTA configuration parameters, as well as managing the prefetcher operation; (ii) an Address Generation Unit (AGU), which solves the proposed pattern description specification and generates its corresponding memory access sequence; and (iii) a prefetch request coalescing unit, entirely similar to the original one from the SM's caches.

When a CTA is issued on a SM by the GPU's workload distribution engine (the GigaThread Engine, in NVIDIA GPUs [16]), its memory access pattern description code is loaded with the kernel code and sent to the stream prefetcher controller, together with the configuration parameters of the CTA (depicted in Fig. 4). Upon their reception, the controller initializes the descriptors with the parameter values and stores them in a dedicated scratchpad memory.

In turn, the proposed stream prefetcher is itself integrated with the conventional GPU cache hierarchy. Hence, the generated memory address requests are coalesced by a dedicated prefetch coalescing unit and sent to the global memory. Moreover, the stream requests issued to the global memory bypass the *L*2 cache banks, to avoid

the introduction of more contention and to not interfere with the coherency policy of the shared cached data. Upon reception of the requested data streams, they are stored in a convenient prefetch buffer, which works in parallel with the $L1$ data cache. The deployment of such a structure results from the fact that the prefetched data are obtained before it is needed by the processing infrastructure and can easily replace data that are being used at the time. As such, it not only allows possible prefetching overheads to be hidden but also avoids filling up the cache memory with prefetch data, which could otherwise incur in premature cache line eviction, and ultimately degrade the overall performance.

Accordingly, the requests to identical memory regions issued either by the stream prefetcher AGU or by miss requests from the $L1$ cache, are merged and registered in the MSHR. This way, requests issued by the cache can be checked, by the stream prefetcher, against the data already stored in the prefetch buffer. In case the required data are present, it is immediately copied to the $L1$ cache and sent to the SM. Otherwise, the miss request is either merged to an outstanding prefetch request or simply registered and sent to the $L2$ cache.

To efficiently generate the memory access pattern described by the adopted descriptor specification, the AGU architecture (depicted in Fig. 4) must execute in the least number of clock cycles (per memory address) as possible, by iterating over each descriptor (described in Eq. 1 and Sect. 3.1). In order to keep the architecture footprint as low as possible, it is based solely on binary adders, divided by three parallel functional blocks (*stride control*, *offset control* and *count control*), and a register bank to store the iteration status. Each block performs one iteration per clock cycle, corresponding to the computation of the current memory address, the multiplication factors for the next iteration and subsequent descriptor state, together with its corresponding control flags.

The stream prefetcher also integrates a dedicated unit responsible for managing the execution of the AGU. Hence, after receiving a given descriptor code, it activates the address generation procedure and the corresponding memory request issuing and coalescing mechanisms. These procedures are performed independently of the remaining prefetcher operations and execute until the completion of the access pattern encoded in the descriptor being solved.

5 Experimental evaluation

The proposed stream prefetching mechanism was integrated in the GPGPU-Sim [3] cycle-accurate simulator (version 3.2.2), whose most recent supported NVIDIA architecture corresponds to the Fermi^{TM} GPU (GTX480) [16]. The adopted simulator configuration is detailed in Table 1, together with the considered subsets of the Rodinia [4] and Polybench [6] benchmark suites. Power consumption was estimated with the GPUWattch [14] tool. All the benchmark application workloads were fully simulated for the baseline and stream prefetching architectures. The obtained results are shown in Fig. 5, with the benchmarks ordered by computational intensity (see Sect. 2).

Table 1 GPGPU-Sim configuration for a NVIDIA Fermi^{TM} architecture model (left) and adopted Rodinia [4] and Polybench [6] benchmark applications and datasets (right)

SIMT core	16 cores, SIMT width=32, 5-Stage pipeline, 1.4GHz
Core Resources	48KB scratchpad, 32768 registers, 32 MSHRs, 1536 threads, 48 warps
*L*1 Cache	32KB/core, 4-way, 128B line, coalescing enabled
Stream Prefetcher	32KB prefetch buffer (per core), 1KB descriptor memory
*L*2 Cache	8 banks, 128 KB/bank, 16-way, 128B line, write-through policy
Scheduling Policy	LRR warp scheduling, round-robin CTA scheduling
Interconnect	32B channel width, 1.4GHz, BW = 350 GB/s per direction
DRAM Model	FR-FCFS Scheduling, 924 MHz, 6 GDDR5 MCs, BW=8Bytes/Cycle
GDDR5 Timing	$t_{CL} = 12$ ns, $t_{RP} = 12$ ns, $t_{rC} = 40$ ns, $t_{RAS} = 28$ ns, $t_{RCD} = 12$ ns, $t_{RRD} = 6$ ns

Application	Benchmark suite	Input size
lud_cuda	Rodinia [4]	256
dwt2d	Rodinia [4]	1024 × 1024
backprop	Rodinia [4]	65,536
srad_v2	Rodinia [4]	2048 × 2048
needle	Rodinia [4]	2048
3DConvolution	Polybench [6]	256 × 256 × 256
gemm	Polybench [6]	512 × 512 × 2
2DConvolution	Polybench [6]	4096 × 4096
gaussian	Rodinia [4]	512
euler3d	Rodinia [4]	97 K
bfs	Rodinia [4]	1 M nodes
bicg	Polybench [6]	16 M
mvt	Polybench [6]	16 M
atax	Polybench [6]	16 M
gesummv	Polybench [6]	16 M

The graph presented in Fig. 5a depicts the attained *L*1 data cache efficiency. As it can be observed, a significant cache hit-rate improvement is obtained (between 34 and 82%), resulting in a clear relation between the performance and cache efficiency improvements, and the computational intensity of each considered application

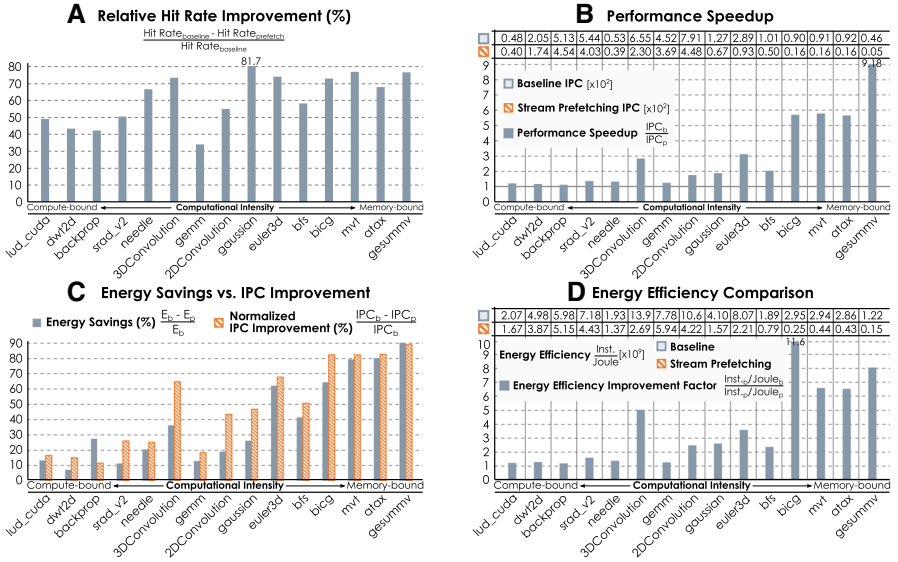


Fig. 5 Relative *L*1 cache hit-rate improvement, resulting performance speedup and energy efficiency for the adopted benchmark set

(compare also with Fig. 1). Moreover, the measured improvements reflect the allied capabilities that are provided by the proposed stream prefetcher to mitigate the cache performance degradation effects of memory-bound applications and complex memory access behaviors. In particular, it shows hit-rate improvements between 68 and 82% for the `euler3d`, `gaussian`, `atax`, `bicg`, `gesummv` and `mvt` memory-bound applications. The exception concerns the `bfs` benchmark, where a smaller improvement is observed (58%), due to its irregular data access nature [4]. Although slightly less effective for the remaining compute-bound applications, the proposed prefetching mechanism still allows a significant average cache hit-rate improvement of 61.3%.

From the graph presented in Fig. 5b, it is possible to ascertain that the proposed stream prefetching mechanism allows performance speedups (IPC change against baseline setup) as high as 9.2× (in memory-bound applications), resulting from an early prefetching of the exact data sequence and from a mitigation of the number of compulsory misses. In general, the attained performance gains range from 2.8× and 3.1×, measured for the `3DConvolution` and `euler3d`, up to the higher speedups of 5.7× and 9.2×, in the `bicg` and `gesummv` applications, respectively. These values show the ability of the proposed stream prefetching mechanism to hide long memory accesses (and mitigate their performance degradation). Moreover, it is shown that even though computationally intensive applications inherently tend to masquerade the impact of the communication infrastructure on the global application performance, they can still achieve a significant performance improvement when aided by the proposed mechanism.

To complete the evaluation of the proposed GPU stream prefetching mechanism, an energy efficiency study was also conducted regarding the adopted set of benchmarks.

From the results presented in Fig. 5c, it is possible to ascertain that the energy consumption reduction is directly related to the attained performance gains (i.e., resulting from the execution time reduction). As a result, the proposed mechanism allows energy consumption reductions from 7 up to 90%, corresponding to the most compute-bound and memory-bound applications, respectively. This is inherently reflected in the considered performance-energy consumption efficiency metric (Instructions/Joule), presented in the table and graph depicted in Fig. 5d. It shows that the prefetching mechanism increases the efficiency of all benchmark applications. As expected from the previously presented performance gains, a major efficiency improvement is observed for memory-bound applications, leading to a maximum of 11.6× efficiency improvement in the `bicg` application. The impact of the prefetching mechanism is also highlighted for the most computational intensive applications, incurring in at lest 1.2× energy efficiency improvements, in the `dwt2d` and `backprop` benchmarks.

The attained cache efficiency and instructions per clock cycle (IPC) improvements support the gains and capabilities of the proposed stream prefetching mechanism. These gains also evidence the fact that an exact description of the kernel memory access sequence allows an accurate and efficient prefetching procedure. Since it does not rely on complex feedback and monitoring mechanisms, consequently eliminating control overheads and prefetching inaccuracies, the proposed mechanism is capable of providing full coverage for deterministic memory access sequences, while still reducing non-negligible waiting times reported in approaches such as [11] and [19]. In fact, adaptive state-of-the-art approaches report prefetching accuracies not greater than 89% [10] and 93.5% [19]. Notwithstanding, while the achieved cache hit-rate improvement is in line with other methods [10,19], the elimination of complex control and monitoring delays, combined with the adopted CTA-level prefetching synchronization, results in higher (3x) performance speedups (on average).

6 Conclusion

To address the limitations of current GPU data cache hierarchies and prefetching schemes, a stream prefetching mechanism, adopting an efficient data-pattern descriptor specification, is presented. The proposed mechanism is able to encode the exact memory access pattern of a given application kernel and is supported by a new low-profile buffered prefetcher that is aggregated to the GPU memory interface. Moreover, it takes advantage of the existing data cache management resources of the GPU, in order to mitigate otherwise added contention and memory traffic. The proposed prefetcher is also able to intercept cache miss memory requests, serving them either with buffered prefetched data or merging them with outstanding prefetch requests in the MSHRs.

The proposed stream prefetching mechanism was integrated in the GPGPU-Sim simulator [3] and an experimental evaluation was conducted for a significant subset of the Rodinia [4] and Polybench [6] benchmark suites. When compared with traditional approaches, the obtained results show that the proposed mechanism is capable of increasing the $L1$ data cache hit rate by an average of 61%, resulting in performance speedups as high as 9.2× and consequent energy efficiency improvements as high as 11.6×.

Acknowledgements This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UID/CEC/50021/2013 and research grant SFRH/BD/100697/2014.

References

1. Amilkanthwar M, Balachandran S (2013) CUPL: A compile-time uncoalesced memory access pattern locator for CUDA. In: Proceedings of the 27th ACM International Conference On Supercomputing. ACM, pp 459–460
2. Arnau JM, Parcerisa JM, Xekalakis P (2012) Boosting mobile GPU performance with a decoupled access/execute fragment processor. ACM SIGARCH Comput Archit News 40(3):84–93
3. Bakhoda A, Yuan GL, Fung WW, Wong H, Aamodt TM (2009) Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 163–174
4. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization (IISWC), pp 44–54
5. Ghosh S, Martonosi M, Malik S (1997) Cache miss equations: An analytical representation of cache misses. In: ACM International Conference on Supercomputing. ACM Press, pp 317–324
6. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: Innovative Parallel Computing (InPar), 2012. IEEE, pp 1–10
7. Grosser T, Groesslinger A, Lengauer C (2012) Polly—performing polyhedral optimizations on a low-level intermediate representation. Parallel Process Lett 22(04):1250010
8. Jia W, Shaw K, Martonosi M (2014) MRPB: Memory request prioritization for massively parallel processors. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 272–283
9. Jia W, Shaw KA, Martonosi M (2012) Characterizing and improving the use of demand-fetched caches in GPUs. In: Proceedings of the 26th ACM International Conference on Supercomputing. ACM, pp 15–24
10. Jog A, Kayiran O, Mishra AK, Kandemir MT, Mutlu O, Iyer R, Das CR (2013) Orchestrated scheduling and prefetching for GPGPUs. ACM SIGARCH Comput Archit News 41(3):332–343
11. Lakshminarayana NB, Kim H (2014) Spare register aware prefetching for graph algorithms on gpus. In: IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp 614–625
12. Lee J, Lakshminarayana NB, Kim H, Vuduc R (2010) Many-thread aware prefetching mechanisms for GPGPU applications. In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 213–224
13. Lee S, Kim K, Koo G, Jeon H, Ro WW, Annavaram M (2015) Warped-compression: enabling power efficient GPUs through register compression. In: 42nd Intl Symposium on Computer Architecture. ACM, pp 502–514
14. Leng J, Hetherington T, ElTantawy A, Gilani S, Kim NS, Aamodt TM, Reddi VJ (2013) GPUWattch: enabling energy optimizations in GPGPUs. ACM SIGARCH Comput Archit News 41(3):487–498
15. Neves N, Tomás P, Roma N (2017) Adaptive in-cache streaming for efficient data management. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(7):2130–2143
16. NVIDIA (2009) NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}. NVIDIA, Santa Clara, Calif, USA
17. NVIDIA (2016) NVIDIA GP100 Pascal Architecture. White paper (Online). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
18. Panda R, Eckert Y, Jayasena N, Kayiran O, Boyer M, John LK (2016) Prefetching techniques for near-memory throughput processors. In: Proceedings of the 2016 International Conference on Supercomputing, ICS '16. ACM, New York, pp. 40:1–40:14
19. Sethia A, Dasika G, Samadi M, Mahlke S (2013) APOGEE: Adaptive prefetching on GPUs for energy efficiency. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. IEEE, pp 73–82
20. Stephenson M, Hari SKS, Lee Y, Ebrahimi E, Johnson DR, Nellans D, O'Connor M, Keckler SW (2015) Flexible software profiling of GPU architectures. In: 42nd International Symposium on Computer Architecture. ACM, pp 185–197

21. Torres Y, Gonzalez-Escribano A, Llanos DR (2011) Understanding the impact of CUDA tuning techniques for Fermi. In: International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp 631–639
22. Wu B, Zhao Z, Zhang EZ, Jiang Y, Shen X (2013) Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. ACM SIGPLAN Not 48(8):57–68
23. Xie X, Liang Y, Wang Y, Sun G, Wang T (2015) Coordinated static and dynamic cache bypassing for GPUs. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 76–88