The Journal of Supercomputing
https://doi.org/10.1007/s11227-022-04621-1

Irregular accesses reorder unit: improving GPGPU memory coalescing for graph-based workloads

Albert Segura[1] · **Jose Maria Arnau**[1] · **Antonio Gonzalez**[1]

Accepted: 27 April 2022

Abstract

GPGPU architectures have become the dominant platform for massively parallel workloads, delivering high performance and energy efficiency for popular applications such as machine learning, computer vision or self-driving cars. However, irregular applications, such as graph processing, fail to fully exploit GPGPU resources due to their divergent memory accesses that saturate the memory hierarchy. To reduce the pressure on the memory subsystem for divergent memory-intensive applications, programmers must take into account SIMT execution model and memory coalescing in GPGPUs, devoting significant efforts in complex optimization techniques. Despite these efforts, we show that irregular graph processing still suffers from low GPGPU performance. We observe that in many irregular applications the mapping of data to threads can be safely changed. In other words, it is possible to relax the strict relationship between thread and data processed to reduce memory divergence. Based on this observation, we propose the Irregular accesses Reorder Unit (IRU), a novel hardware extension tightly integrated in the GPGPU pipeline. The IRU reorders data processed by the threads on irregular accesses to improve memory coalescing, i.e., it tries to assign data elements to threads as to produce coalesced accesses in SIMT groups. Furthermore, the IRU is capable of filtering and merging duplicated accesses, significantly reducing the workload. Programmers can easily utilize the IRU with a simple API, or let the compiler issue instructions from our extended ISA. We evaluate our proposal for state-of-the-art graph-based algorithms and a wide selection of applications. Results show that the IRU achieves a memory coalescing improvement of 1.32x and a 46% reduction in the overall traffic in the memory hierarchy, which results in 1.33x speedup and 13% energy savings on average, while incurring in a small 5.6% area overhead.

Keywords GPGPU · Graph processing · Parallel architectures · Computer architecture

✉ Albert Segura
asegura@ac.upc.edu

Extended author information available on the last page of the article

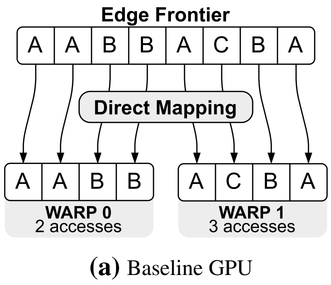
Published online: 18 July 2022

1 Introduction

Since its popularization over the last decade, GPGPU architectures have enabled a broad domain of new applications by boosting linear algebra computations [1, 2], empowering Big Data analytics [3] and deploying Machine Learning [4] in numerous fields such as speech recognition [5], image processing [6] and self-driving cars [7]. GPGPU architectures excel at processing highly parallel throughput oriented applications, which exhibit regular execution and memory access patterns. However, applications that show irregular memory accesses or branch divergence suffer from severe underutilization of GPGPU's functional units [8]. Graph processing algorithms are a popular example of irregular applications. Although graph-processing can potentially benefit from highly parallel architectures, they process unstructured and irregular data, which results in sparse and unpredictable memory access patterns [9]. In addition, graph processing shows extremely low computation to memory access ratio [10], which further hinders GPGPU efficiency.

GPGPU programming models such as CUDA employ threads to exploit parallelism, each thread processing its own set of data while synchronizing with the rest to perform complex behaviors determined by the algorithm. The GPGPU pipeline handles the execution of warps, i.e., groups of threads in lock-step execution. The number of threads and the ability to coalesce the memory accesses within a warp are some of the key factors that determine the utilization of the GPU resources. The simplest way to exploit parallelism is to instantiate as many threads as data elements to process and directly assign each element to a given thread, as seen in Fig. 1a. For a regular program, this assignment is highly effective at achieving high utilization of resources without inefficiencies (e.g., vector addition, where each thread processes consecutive data in memory achieving regular behavior). For programs exhibiting irregular memory accesses, this simple assignment might cause utilization degradation, as the GPU is unable to achieve high memory coalescing in a warp, resulting in poor data locality (e.g., graph processing, where each thread processes a given node of the graph and has to fetch its adjacent ones).

To mitigate the aforementioned problems, GPGPU algorithms have to carefully consider the underlying hardware and adapt the algorithm to minimize



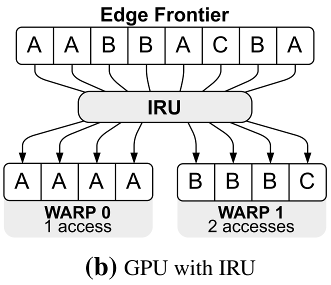


Fig. 1 Memory Coalescing improvement achieved by employing the IRU (**b**) to reorder data elements that generate irregular accesses versus a Baseline GPU (**a**) execution. Assuming warp size of four threads for the sake of simplicity

branch divergence and improve memory coalescing, among other performance optimizations [11, 12]. Graph algorithms employ many such techniques, such as scan algorithms [13] which are leveraged for data compaction [14] that gathers data to be accessed sparsely into a compacted data array, improving locality and memory coalescing. These techniques shift programmers effort from the algorithm to a hardware conscious programming requiring sound knowledge of the microarchitecture, significantly increasing development time while hampering code portability. Another approach is to cleverly organize threads [15] in a thread block to collaborate among themselves with the processing of unbalanced data entries, which leads to reduced branch divergence and increased overall resources utilization. While these techniques ameliorate the shortcomings of graph processing and irregular access applications on GPGPU architectures, they clearly shift the programmers effort from the algorithm to a hardware conscious programming requiring sound knowledge of it and hampering code portability.

We claim that GPGPU programming models impose restrictions that hinder full resource utilization of irregular applications for several reasons. First, irregular programs such as graph processing algorithms consist of sparse and irregular memory accesses which have poor data locality and result in low memory coalescing, producing intra-warp memory divergence and significantly reducing GPU efficiency. Second, these issues are hard to improve without significant programmer effort to modify algorithms and data structures in order to better utilize the underlying hardware, which in some cases may not even be feasible and thus effectively limit the achievable performance. Ultimately, the programmer has to take into consideration ways to rearrange the data or change the mapping of data elements to threads to achieve better memory coalescing and higher GPU utilization, even if the relation of which threads process what data might not even be a restriction imposed by the algorithm, since the threads are primarily the means to expose parallelism. Since GPGPU architectures and programming models are not designed to efficiently support sparse irregular programs, we propose to extend the GPU architecture to improve these workloads with a set of new instructions and their corresponding hardware support. We call this hardware the Irregular accesses Reorder Unit (IRU). The IRU is a small unit tightly integrated in the GPU, that is accessible through a set of new ISA instructions which can be used by the compiler or the programmer through a simple high-level API.

Our key idea is to relax the strict relation between a thread and the data that it processes. This allows the IRU to reorder the data serviced to the threads, i.e., to decide at run-time the mapping between threads and data elements to largely improve memory coalescing. Figure 1 shows conceptually how the IRU assigns data to the threads and achieves an improvement in memory coalescing against the baseline GPU. The IRU mapping improves the effectiveness of the memory coalescing hardware and the L1 data cache, as it results in better coalescing and locality, with subsequent improvements in the entire memory hierarchy, resulting in higher GPU utilization for irregular applications. In addition, the IRU performs simple preprocessing on the data (i.e., filtering repeated elements), which reduces workload and allows for better utilization and further performance and energy improvements. In conclusion, the IRU optimizes irregular accesses requiring minimal support from programmers.

This paper focuses on improving the performance of irregular applications, such as graph processing, on GPGPU architectures. Its main contributions are the following:

- We characterize the degree of memory coalescing and GPU utilization of modern graph-based applications. Our analysis shows that memory coalescing can be as high as 4 accesses per warp and GPU utilization as low as 13.5%.
- We propose the IRU, a novel hardware unit integrated in the GPGPU architecture enabling improved performance of irregular accesses by reordering data serviced to each thread. We further extend the IRU to filter repeated elements, largely reducing GPU redundant workload for graph applications.
- We propose an ISA extension and API showing how modern graph-based applications can easily leverage the IRU.
- The GPU architecture with our IRU improves memory coalescing by a factor of 1.32x and reduces NoC traffic by 46%, which result in 1.33x speedup and 13% energy savings on average for a diverse set of graph-based applications. The IRU represents a small area overhead of 5.6%.

The remainder of this paper is organized as follows. Section 2 reviews the challenges of irregular graph processing on GPGPU architectures. Section 3 presents the architecture of the IRU, and Sect. 4 describes its API and usage for graph applications. Section 5 describes the methodology, and Sect. 6 presents the evaluation. Section 7 reviews relevant related work, and, finally, Sect. 8 sums up the main conclusions.

2 Irregular applications on GPU architectures

GPGPU architectures are tailored for compute intensive applications that feature regular execution and regular memory access patterns. GPU's high IPC is enabled by its Single-Instruction, Multiple-Threads (SIMT) pipeline, leveraging the advantage of decoding a single instruction for multiple threads, each operating on different data. The threads in a warp execute in a lock-step manner and, hence, to fully utilize the Execution Units (EU) applications must exhibit regular access patterns and control flow. Furthermore, to sustain high IPC, significant memory bandwidth is required which is accomplished with high Memory-Level Parallelism (MLP) leveraging warp-level coalescing and concurrent execution of many threads, increasing memory bandwidth at the expenses of increased latency.

On the other hand, for applications that show irregular behavior with unpredictable memory access patterns, GPGPU architectures are unable to provide enough memory bandwidth due to a huge portion of the threads generating uncoalesced accesses, which further hampers performance and results in low utilization of the EUs due to increased stalls. In the worst case, a warp-level memory instruction requires 32 memory accesses (assuming warp size of 32 threads), as each thread may access a different cache line, whereas a perfectly coalesced warp-level memory instruction only requires one memory request (i.e., in case all the threads access the same cache line[1]). Therefore, an irregular application may increase the requests to the memory subsystem by 32x compared to an application with regular access patterns and perfect memory coalescing.

Not surprisingly, irregular applications increase the utilization of the LD/ST unit, the latency of memory instructions and the pressure on the L1 and the whole memory hierarchy. In addition, every warp instruction requires more resources to handle misses, such as miss status holding registers (MSHRs) and entries in the miss queue, a problem aggravated by GPUs small capacity ratio of cache lines per thread compared to CPUs. All these factors significantly increase the contention and conflict/capacity misses on the L1. Finally, the interconnection traffic also increases, L2 suffers from similar problems to the L1, and main memory accesses increase as a consequence of increased L2 misses.

Significant changes have to be applied to an algorithm and its data structures in order to reduce irregular accesses' overheads, and improve GPU efficiency. Generic approaches include the use of the shared memory in the Streaming Multiprocessors (SM) of the GPU, providing reduced latency and banked accesses of uncoalesced requests. Other approaches favor merging kernels, avoiding redundant memory requests at the cost of higher register usage. Graph algorithms use techniques such as data compaction [14], which reduce sparse accesses and improve locality by gathering sparse data in a compacted data array, as well as load balancing techniques [16] that leverage collaborating threads which reduce branch and memory divergence.

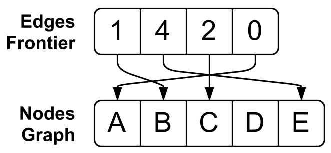
Overall, irregular applications benefit from the high performance delivered by the massive parallelism of GPU architectures, but the architecture has significant bottlenecks that result in low performance for irregular algorithms. Significant programmer effort, code complexity and underlying hardware knowledge are required to create efficient GPU code for irregular applications such as graph processing algorithms.

2.1 Graph processing on GPGPU architectures

Many problems in Machine Learning [17, 18] and Data Analytics [19] are modeled using graphs, which represent relationships between the elements on a set of data. GPGPU architectures enable fast parallel exploration and processing of the nodes and connections (i.e., edges) of a graph. Nonetheless, graph exploration is low-computation intensive [10], unstructured and irregular [20, 21] with sparse, irregular and highly unpredictable access patterns due to the irregular nature of the relationships expressed in a graph.

A typical GPGPU graph processing algorithm starts in a given node and moves to adjacent nodes by traversing, or processing, that node edges. At this point, a new frontier (i.e., set of nodes or edges) is ready to be explored continuing this process

[1] If a sectored cache is used, perfect coalescing is only achieve if all the threads in a warp access the same sector. Note that if multiple sectors of a line are accessed, then multiple requests will be generated even if all the threads access the same cache line.



```
for i in range(0, N):
  idx = edges_frontier[i]
  out[i] = nodes_graph[idx]
```

Fig. 2 The graph edges frontier produces irregular memory requests when accessing the nodes data in the graph. In a GPGPU, each thread may process one of the N elements in the edges frontier, i.e., perform one iteration of the loop. In this case, the access to *edges_frontier* shows high memory coalescing, as consecutive threads access consecutive memory locations. However, the access to *nodes_graph* array may result in high memory divergence depending on the indices

iteratively until the whole connected graph is explored, or until the algorithm dictates it. Figure 2 shows how this process unfolds in a given iteration; each element of the edges frontier array (i.e., indices) points to the position to access in the nodes array to fetch for the next frontier data and continue the graph exploration. The pseudo-code shows the type of irregular access performed, which is an intrinsic part of graph exploration algorithms and a cause of the previously mentioned memory divergence. In this work, we focus on common graph algorithms, in particular Breadth-First Search (BFS) [15], Single-Source Shortest Paths (SSSP) [22] and PageRank (PR) [23].

GPGPU graph processing leverages many strategies to improve performance. First, data structures that efficiently represent the graph data in a compact manner using the Compressed Sparse Row (CSR) [24] format. Second, to cut down on sparse accesses, stream compaction algorithms [14] are used to gather data in contiguous memory improving data locality and coalescing. Finally, load balancing techniques [15] are used to leverage the threads in warps and thread blocks to cooperatively process data from the more processing demanding nodes, since the irregular graph connectivity of the nodes leads to nodes that largely differ in the number of edges. Although these techniques improve GPGPU efficiency for graph processing, significant changes are required to implement these optimizations and reduce the GPGPU architecture bottlenecks for irregular applications. Despite all these efforts, we observe that modern graph applications experience significantly low memory coalescing of 4 accesses per warp, leading to a low 13.5% utilization of GPU resources. In the next section, we present a novel hardware unit that improves memory coalescing and GPU performance for irregular workloads, while requiring minimal changes in the applications.

3 Irregular accesses reorder unit

In this section, we introduce the Irregular accesses Reorder Unit (IRU), which improves performance of irregular workloads such as graph applications on GPGPU architectures. Low GPGPU performance for graph processing is mainly due to the uncoalesced memory accesses that result in large memory traffic and put significant pressure on the memory hierarchy. Our proposal improves GPGPU performance for graph-processing by assigning data elements (nodes/edges) that produce coalesced

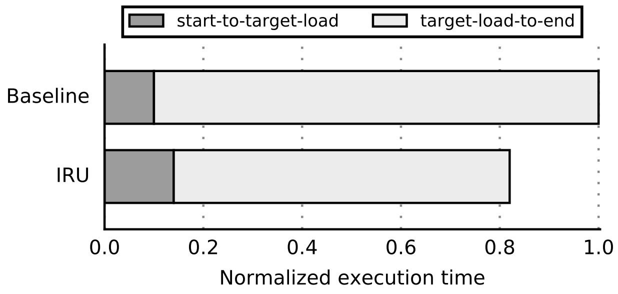
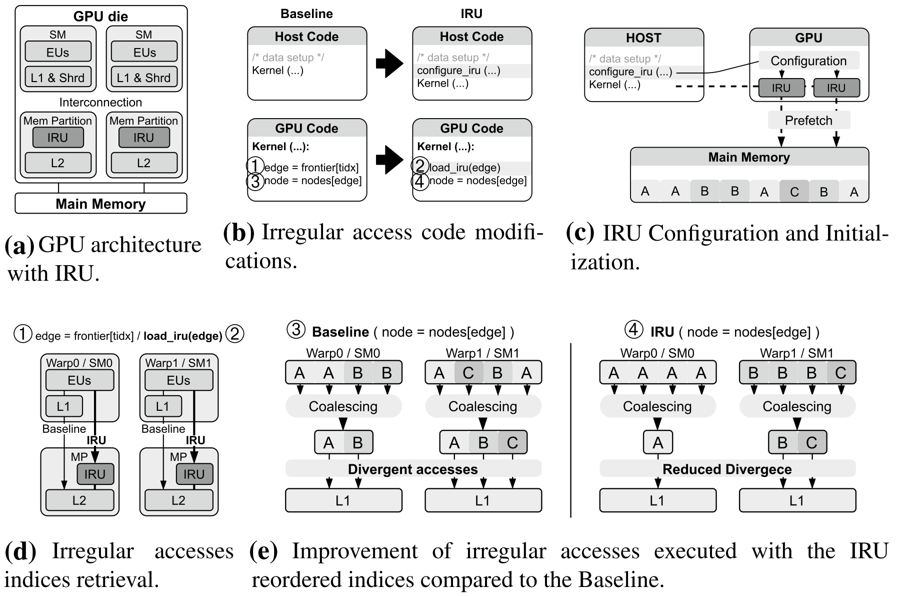


Fig. 3 Warp average normalized execution with and without IRU. The dark bar indicates execution time until the target load is serviced, and the light bar from service to finalization. Processing a load instruction with the IRU is slower as it has to reorder data elements to identify indices that target the same cache line. However, once the indices are sent to the SMs, the remaining execution is faster as subsequent memory accesses used the IRU-prepared indices that result in higher memory coalescing



(a) GPU architecture with IRU.

(b) Irregular access code modifications.

(c) IRU Configuration and Initialization.

(d) Irregular accesses indices retrieval.

(e) Improvement of irregular accesses executed with the IRU reordered indices compared to the Baseline.

Fig. 4 IRU integration with the GPU at different levels: architectural (**a**), program model (**b**) and execution (**c**–**e**). The execution shows how the program (**b**) works on the Baseline and the IRU-enhanced GPU with two warps and data from Fig. 1

memory accesses to the threads in the same warp. This dynamic reordering of elements is done in hardware, and it can be easily used by programmers, that only have to indicate when it is safe to change the mapping of data elements to threads.

In this paper, we propose to extend the GPGPU with the IRU to reduce the overheads caused by irregular accesses. The IRU is a compact and efficient hardware unit integrated into the Memory Partition (MP) of the GPU architecture as shown in Fig. 4a, which incurs in very small energy and area overheads. The IRU leverages the observation that GPU programs employ threads to convey parallelism; being in many cases independent of the data that they process. The main goal of the IRU is to process, reorder and redistribute the indices used to perform irregular memory accesses. The reordering collocates indices that access the same memory block and services them to a requesting warp, reducing the memory divergence of irregular accesses. In turn, the improved memory coalescing reduces congestion of the resources of the LD/ST unit, L1, interconnection, L2 and main memory, significantly reducing the pressure on the memory subsystem. In addition, the reordering is performed across all the indices accessed by all the SMs, and hence, collocating irregular accesses potentially gathers data obtained by irregular accesses in a single or fewer SMs, thus further reducing interconnection traffic and L1 data thrashing.

Figure 3 shows the average normalized execution of a warp in a baseline GPU against one with the IRU. The dark bar indicates the execution time until the load processed and reordered by the IRU is serviced, while the light bar shows the normalized time until finalization. As it can be seen, processing a load instruction through the IRU is slower as it has to reorder the data elements to identify indices that target the same cache line. Furthermore, it also identifies and filters duplicated elements. For this reason, the *start-to-target-load* time in Fig. 3 is larger with the IRU than in the baseline GPU. However, once the indices are reordered and sent to the warps, the remaining execution is significantly faster as memory coalescing is largely improved for the subsequent memory accesses. Therefore, the low overhead incurred by the IRU servicing the load is effectively offset by the performance gain achieved from the reduction in the overheads due to the memory divergence.

The IRU processes the indices of a target irregular instruction, with the objective to improve its coalescing. Additionally, the elements processed can contain more data than just the indices, as mandated by the API described in Sect. 4. While these data are not used for the IRU coalescing logic, it is responsible to fetch and send these additional data to the SM.

3.1 GPU integration

The IRU integration into the GPU is covered in Fig. 4, showing architectural 4a, programming 4b and execution 4c–d integration. The execution shows how the Baseline and the IRU-modified GPU programs in Fig. 4b operate with the two warps and data from Fig. 1. The Baseline program performs a regular access ② to gather indices that are then used for an irregular access. The IRU-modified code performs the same operation but using the IRU hardware with the *load_iru* operation ②, which is part of the IRU API presented in Sect. 4. The baseline code is executed by the GPU as follows. First, the two warps retrieve the indices performing regular accesses to the L1, i.e., consecutive threads in a warp access consecutive memory addresses. Afterward, Fig. 4e shows how they perform irregular accesses to the L1 with the retrieved indices which, due to the high divergence, result in many memory requests ③.

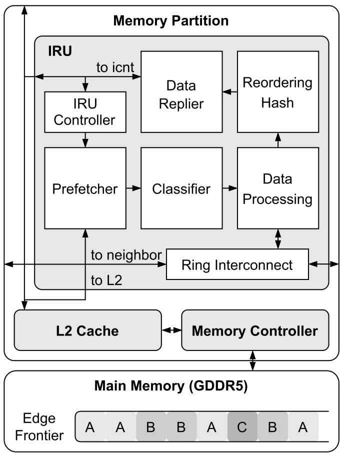
In contrast, the IRU program first introduces a configuration step performed on the host, shown in Fig. 4c, that provides data of the irregular accesses to optimize. The configuration required for this program consists of the base address and data type of the irregular accessed data, and the indices array and total number of irregular accesses. Further, IRU capabilities are enabled and used with optional parameters employed on overloaded functions, reviewed in Sect. 4. Next, when the kernel execution starts, the IRU triggers the prefetching of the indices from L2 and memory, which are then automatically reordered in the IRU hash. The IRU activity is overlapped with the execution of the kernel and disabled when all the data are processed. Furthermore, the IRU is disabled for kernels that do not require reordering indices.

Regular execution proceeds until encountering the *load_iru* operation, at which point the warps retrieve the indices performing requests directly to the IRU, bypassing the L1 as seen in Fig. 4d. The IRU replies with reordered indices either instantly, if 32 indices that target the same cache line are available, or otherwise after a timeout to avoid starvation. In case the timeout is triggered, the 32 indices will not be collocated to the same cache line, but the IRU will do the best attempt to provide indices that result in the lowest memory divergence with the available elements. Finally, the warps perform the irregular access that was the target of the optimization ④. This access is performed with the IRU reordered indices which achieves reduced divergence, performing less accesses than the baseline program, as depicted in Fig. 4e. Note that we assume a warp size of 4 threads in Fig. 4e for the sake of simplicity, but we use warp size of 32 for our experimental evaluation.

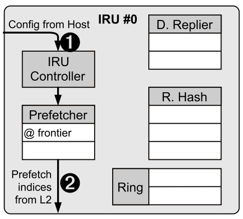
The IRU is first configured by the programmer with a host function, described in Sect. 4, that provides the IRU information about the data that has to be reordered. Once configured, the kernel is launched and the IRU begins its operation prefetching the required data and reordering the elements according to the configuration, with the aim of maximizing memory coalescing. IRU operation is autonomous, and no further programmer intervention is required to reorder the elements. The SM kernel retrieves the reordered indices used by the irregular accesses using the API described in Sect. 4, requiring minimal changes to the code.

3.2 Hardware overview and processing

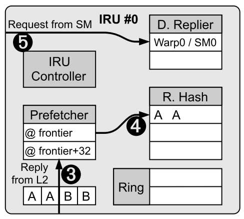
The hardware architecture of the IRU is shown in Fig. 5a. The main purpose of the IRU, which is to reorder indices to improve memory coalescing, is accomplished with the use of a hash table located inside the *Reordering Hash* block. The IRU is integrated inside the memory partitions of the GPGPU architecture, i.e., together with the L2 cache partition, the atomic operations unit and the memory controller. GPUs include multiple memory partitions, in our proposal each partition will contain an instance of the IRU. Instead of having multiple private hash tables, there is a single logical hash table partitioned among the IRUs. This motivates the inclusion of a ring interconnection between the IRUs to forward the data to the corresponding partition of the logical hash table. We have observed that the degree of memory coalescing is significantly affected if each IRU hash table is private and separated; which would constrain IRUs reordering scope to data from a single memory partition. Finally, requests are issued to the L2 to exploit data locality among kernel



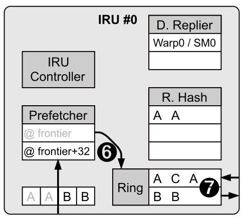
(a) Overview of the IRU internal pipelined architecture.



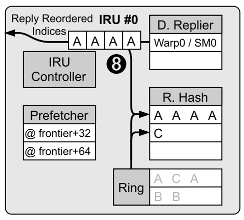
(b) Configuration and Prefetch.



(c) Data and Request retrieval.



(d) Ring interconnection.



(e) Requests reply.

Fig. 5 Architecture and the internal processing performed by the IRU. The indices in memory (from Fig. 1) are processed by two IRU partitions (IRU 0 shown), which are later replied to a request coming from Warp 0 in SM 0

executions. Alternatively, requests can be configured to bypass L2, which could be beneficial for streaming kernels.

The overall internal processing of the IRU is shown in Fig. 5. The figure covers a general overview of the internal IRU architecture and the detailed step by step working of the most relevant components of the IRU covering: configuration and prefetching (5b), data and requests retrieval (5c), ring interconnection interaction (5d) and requests reply (5e).

3.2.1 Prefetching and data processing

The *IRU Controller* is initialized by the Host by executing the *configure_iru* function with the corresponding data ❶. Later the *Prefetcher* uses this data to determine the addresses to prefetch when the GPU kernel starts execution ❷. The *Prefetcher* issues a limited number of on-the-fly prefetch requests to avoid saturating memory bandwidth. Each IRU only prefetches information from its corresponding memory partition. In Fig. 5, the first four elements from main memory are fetched by IRU 0, while the next four by IRU 1. When a reply comes back, the retrieved data are stored in a FIFO queue to be later processed.

Afterward, the *Classifier* block processes the prefetched data ❸ by splitting it into smaller FIFO queues, with a throughput of one element per cycle on each queue. The smaller FIFO queues contain the elements that will be inserted in the hash or forwarded through the ring. A hashing function of the element is used to determine which hash table entry it is mapped to and, therefore, if it will access a local bank or must be sent through the ring. Finally, the *Data Processing* block retrieves elements from both the smaller FIFO queues and the ring, prioritizing the latter, and forwards them to the ring or inserts them into the local hash table ❹. In Fig. 5e, the elements labeled *A* are inserted into the local hash table, as they are determined to access the same memory block.

Meanwhile, requests from the SMs can be received at any time which are then processed by the *Data Replier* ❺. This request originates directly from the SM (i.e., bypassing the L1) and are generated by the extended ISA *load_iru* operations, that are responsible to retrieve the IRU processed data. Their information is stored until enough data are available to satisfy the request or until a timeout is reached.

3.2.2 Ring and data reply

Due to the partition of the reordering hash table, the hash function of the elements fetched from a memory partition can require that element to be inserted in another IRU partition. The *Ring Interconnection* allows to receive and send elements to the neighbor partitions at every cycle. In Fig. 5d, the elements labeled *B* are determined to correspond to another IRU partition and so are inserted in the ring ❻. Meanwhile, data from the neighbor partition are received (indices *A* and *C* correspond to IRU 0) ❼.

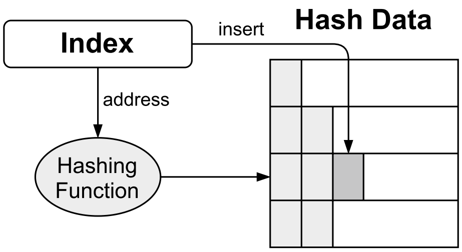
Lastly, the elements corresponding to this IRU partition are gathered from the ring and inserted into the reordering hash table. When the *Data Replier* detects a hash entry that is complete, or enough data are available to reply a request, the oldest request is replied back to the SMs with that entry's reordered elements ❽, and the data are evicted from the hash table. The data used for the reply (four *A*) are the indices used for the irregular access being optimized. Additionally, more data might be processed per element, requiring at most two replies to be issued, as some algorithms require extra data associated with each index.

Additionally, a timeout is employed to avoid excessively delaying a request. Once the timeout is reached, it then fetches data from the hash table with the best coalesced data entry present and sends the response once enough data to satisfy the request is retrieved, effectively trading-off worse coalescing for lower latency. Furthermore, simple control logic is added to the SM and IRU partitions to handle balancing issues (i.e between request and entries ready). Each SM distributes the requests evenly across the different IRUs in the memory partitions, and requests can be replied by IRU partitions other than the original. Finally, when no more data are left to be inserted into the IRU, the *Data Replier* replies to the SM by intelligently merging the remaining hash entries.

3.3 Reordering hash

The *Reordering Hash* contains a physical partition of the global logical hash, which is direct mapped and multi-banked. Each entry holds up to 32 elements that are inserted into the entry in subsequent locations at every hash insertion, as depicted in Fig. 6. Furthermore, the hash function key that points to an entry is generated from the value being inserted into the hash table entry. The computation of the hash

Fig. 6 Hash table insertion diagram showcasing how the element is used for the hashing function and is stored in the hash table data



function collocates in a single hash table entry the elements that will generate memory fetches that target the same memory block, which provides the memory coalescing improvement achieved with the IRU.

Unlike a regular hash table, an insertion allows to merge elements into a hash entry even if the tag does not match. The inherent drawback of this decision is that the elements that a hash table entry collocates might actually not access the same memory block, and thus, the memory coalescing that it can achieve will not be optimal. Nonetheless, this design decision largely reduces hardware complexity. Furthermore, a good dispersion hash function and properly sized hash tables limit the amount of conflicts and the effects on memory coalescing. Ultimately, when an entry is completely filled with 32 elements, no more data can be inserted to it. At this point, it has 32 collocated elements that potentially will access the same memory block when the program uses them to perform an irregular access, unless there were conflicts. Note that some of these conflicting elements might collocate among themselves, thus not severely impairing memory coalescing.

Some API operations described in Sect. 4 require additional comparators or adders to be used in a hash table insertion. The additional data that the elements might have are processed by this hardware, which effectively merges or filters an element present in the hash table with the one being inserted. Since these operations will filter out elements, some threads that requested data will not receive any element, which is handled by the *Data Replier* and exposed to the programmer with the API.

4 IRU programmability

Ease of programmability is a crucial aspect to write efficient parallel programs, a reason for which toolkits such as CUDA are very successful. As described in Sect. 2, efficient irregular programs require complex optimization techniques. The IRU has been designed to be easily programmable, so existing codes can exploit this new hardware unit with minimal changes. The IRU extends the GPGPU ISA to support memory load operations that fetch data from the IRU, which require small changes to the pipeline to decode these instructions, in addition to minor changes to the LD/ST unit to route these requests to the IRU. To avoid directly using assembly instructions, we provide a simple API with functions that can be called from CUDA kernels. Furthermore, since the changes to the code are minimal, a compiler that

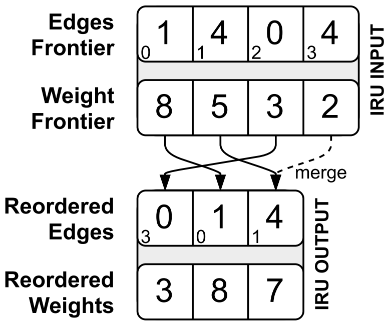


Fig. 7 IRU processing of two arrays with filtering enabled. The edges frontier represents the array of indices, while the weight frontier is the secondary array. Filtering is an additional operation of the IRU that can be enabled to remove duplicated elements in order to reduce workload

supports the ISA extensions can issue the new instructions when appropriate, freeing the programmer from performing the optimization effort and delivering more code for irregular applications.

IRU's main optimization is the reordering of indices fetched from memory that are later used for irregular accesses. This optimization is based on the premise that the assignment of data to the threads can be safely changed, i.e., each data element (e.g., node/edge in a graph application) can be processed by any thread. Consequently, to be able to correctly utilize the IRU for this optimization, the programmer has to guarantee that the reordering can be applied correctly. The API provides additional functionality used to indicate when it is safe to replace a regular load by an IRU load instruction.

The baseline functionality provided by the API and IRU hardware supports reordering of an array of 24-bit indices. Additionally, a secondary 32-bit array can be processed simultaneously, yet the reordering is based on the indices array as to improve the coalescing achieved when performing an irregular access. The data (i.e., index and entry in the secondary array) provided to the threads is reordered applying the same reordering to both indices and secondary array, maintaining the original pair of index and secondary data. Figure 7 shows how the input data, first two rows, are reordered in the output data, last two rows. The reordering is based on the array of indices, the edge frontier, and every edge is kept with its corresponding weight. This secondary array can be used to process attributes or extra data of the elements being processed. It might be the case that more than a single additional array has to be processed in some application. In this case, the reordering operation can return in which position in the original array the reordered element was located. This position can be used to fetch any additional attributes required from multiple arrays.

Graph-based algorithms process many nodes and edges in parallel. Since it is common that several edges lead to the same destination node, many duplicated nodes may appear in the node frontier, producing redundant work in subsequent iterations of the algorithm. This additional work is usually benign as the program implements filtering techniques, which are effective yet computationally costly due to synchronization requirements. To help the programmer remove this additional workload, the IRU is extended to provide filtering or merging of elements (i.e., pair of index and attribute). The IRU can easily detect duplicated indices that are processed simultaneously, and so it can remove them or might perform some operation to merge both

```
void configure_iru (
  addr_t target_array,
  size_t target_array_data_type_size,
  addr_t indices_array, addr_t secondary_array,
  size_t number_elements, filter_op_t filter_op );

__device__ bool load_iru (
  addr_t &indices_array, addr_t &secondary_array,
  uint32_t &position );
```

Fig. 8 IRU API function declarations

elements. The operations supported by the IRU are integer comparison and floating point addition. Figure 7 shows the merging of two indices into one on the output data by adding their attributes in the secondary array. Filtering out elements causes some threads to not receive any data, and so we extend the API to indicate if a given thread's data have been filtered out. IRU groups the disabled threads in warps rather than distributing them across many warps, this approach allows to minimize branch divergence, remove redundant work and improve performance.

The IRU API, shown in Fig. 8, provides two main functions: *configure_iru*, used from the host to configure the IRU, and *load_iru*, used inside the CUDA kernel to retrieve reordered data from the IRU. At the start of kernel execution, the *configure_iru* function is called to provide all the parameters of the data that will be processed. The required parameters are: target array base address and data type width, both parameters used to configure the offset to be applied to the indices as to compute the coalescing required; the indices array is required too, which is the main data reordered; and finally, the number of elements in the indices array. Optional parameters include the additional secondary array, reordered together with the indices array, and the optional filtering operation performed. The memory load operation replaces regular load instructions, retrieves the original position of the indices and indicates if a thread is disabled.

4.1 IRU enabled graph applications

The previously described API enables the instrumentation of state-of-the-art graph-based algorithms such as BFS, SSSP and PR. Although we use push graph implementations, the IRU is not specifically targeting push or pull. The ease of use of our API allows very simple instrumentation an minimal code changes while providing efficient memory coalescing improvements. The following examples show how *load_iru* can be used within GPGPU kernels to easily replace existing code.

The basic functionality of the IRU is a good fit for the BFS algorithm as illustrated in Fig. 9. The indices found in the *edge_frontier* array are used to access the *label* array, resulting in irregular memory accesses and poor memory coalescing. The programmer can easily replace the previous instruction with the *load_iru* operation to obtain the indices in such a way that memory coalescing is improved and thus overall performance increases.

```
1  __global__ void BFS_Contract (...) {
2    int pos = blockDim.x * blockIdx.x + threadIdx.x;
3    if (pos < number_elements) {
4      int edge;
5  #ifdef NOT_INSTRUMENTED
6      edge = edge_frontier[pos];
7  #elif USE_IRU
8      load_iru(edge);
9  #endif
10     label[edge] = distance;
11   }
12 }
```

Fig. 9 Instrumentation of a BFS Kernel using the IRU

```
1  __global__ void SSSP_Compaction (...) {
2    int pos = blockDim.x * blockIdx.x + threadIdx.x;
3    if (pos < number_elements) {
4      int edge, weight;
5  #ifdef NOT_INSTRUMENTED
6      edge = edge_frontier[pos];
7      weight = weight_frontier[pos];
8  #elif USE_IRU
9      load_iru(edge, weight, pos);
10 #endif
11     int old = atomicMin(&label[edge], weight);
12     if (old > weight)
13       lookup[edge] = pos;
14   }
15 }
```

Fig. 10 Instrumentation of an SSSP Kernel using the IRU

The SSSP algorithm processes additional data per element; each edge has an associated weight value. Figure 10 shows how *load_iru* can handle the use of an additional array, while also retrieving the original position of the reordered element in the *pos* variable. Note that the algorithm requires the *pos* variable to be correctly updated with the reordered element in line 17, which is easily accomplished with our API extension.

Finally, the PageRank kernel shown in Fig. 11 performs additions of the elements' weights into the *label* array. Utilizing the filtering/merge functionality of the IRU, an initial addition can be performed while the elements are being processed in the IRU, which allows to disable merged out threads. The *load_iru* function returns whether or not the thread has a valid element or if it has been merged out; the value

```
__global__ void PR_Contract (...) {
  int pos = blockDim.x * blockIdx.x + threadIdx.x;
  if (pos < number_elements) {
    int edge; float weight;
    bool active_thread = true;
#ifdef NOT_INSTRUMENTED
    edge = edge_frontier[pos];
    weight = weight_frontier[pos];
#elif USE_IRU
    active_thread = load_iru(edge, weight);
#endif
    if (active_thread)
      atomicAdd(&label[edge], weight);
  }
}
```

Fig. 11 Instrumentation of a PageRank Kernel using the IRU

in a retrieved element's *weight* has the sum of those *weight* of the same *edge*. Note that the filtering is not complete as it merges only elements found concurrently on the IRU, yet it manages to filter a significant amount of duplicated elements. Overall, this extension allows reducing the workload of the kernel, in this case, reducing the number of *atomicAdd* required.

5 Evaluation methodology

We have implemented the IRU architecture in GPGPU-Sim 3.2 [25]. We extended the memory partitions in GPGPU-Sim to accurately model IRU's hardware as shown in Fig. 5a. Furthermore, we extended the Streaming Multiprocessors (SMs) to support our new instructions.

Each partition of the IRU uses a 2 KB FIFO to buffer warp requests and 1.7 KB for the prefetching buffer (8 on-the-fly prefetches). A buffer of 1.2 KB is used for the Classifier block to determine the data destination. The ring requires a total of 2.8 KB buffering. The main component of the IRU is the direct-mapped hash table with 1024 sets, split in 4 physical partitions. Each partition is 2-way banked, holding 256 sets, requiring 80 KB of total storage, significantly smaller than the 512 KB of the L2 partition. Table 1 summarizes the components of an IRU partition. Since the IRU is mostly comprised of SRAM elements without complex execution units, we model area and energy using CACTI [26] with a node technology of 32 nm.

GPGPU performance is modeled with GPGPU-Sim 3.2 [25], energy consumption and area with GPUWattch [27], both conform a state-of-the-art GPGPU simulator system with validated performance and energy consumption against real GPU hardware. Both simulators configured with the parameters shown in Table 2 to model an NVIDIA GTX 980. To evaluate our proposal, we use state-of-the-art GPGPU implementations

Table 1 IRU hardware requirements per partition

Component	Requirements (KB)
Requests Buffer	2
Prefetcher Buffer	1.7
Classifier Buffer	1.2
Ring Buffer	2.8
Hash Data	80

Table 2 Parameters employed in the experiments to model a GTX 980 in GPGPU-Sim

Characteristic	Configuration
GPU, Frequency	NVIDIA GTX 980, 1.27GHz
Streaming Multiproc.	16 (2048 threads), Maxwell
SM Functional Units	128 EUs, 1 LD/ST per SM
SM Issue Schedulers	4 Warp Schedulers per SM
L1 data cache	32 KB, 4-assoc, 128 B lines
L2 data cache	2 MB, 8-assoc, 128 B lines
Memory Partitions	4 (4 channel GDDR5)
Main Memory	4 GB GDDR5, 224 GB/s

Table 3 Benchmark graph datasets

Graph name	Description	Nodes (10^3)	Edges (10^6)	Avg. degree
ca [28]	California road network	710	3.48	9.8
cond [28]	Collaboration network, arxiv.org	40	0.35	17.4
delaunay [29]	Delaunay triangulation	524	3.4	12
human [28]	Human gene regulatory network	22	24.6	2214
kron [29]	Graph500, Synthetic Graph	262	21	156
msdoor [28]	Mesh of 3D object	415	20.2	97.3

of BFS [15], SSSP [22], and PageRank [23] graph algorithms. We run these graph processing algorithms with datasets representative of different application domains with varied sizes, characteristics and degrees of connectivity, shown in Table 3 and collected from well-known repositories of research graph datasets [28, 29].

6 Experimental results

In this section, we evaluate the performance and energy efficiency of our IRU hardware presented in Sect. 3. More specifically, we analyze how the memory hierarchy contention is reduced, the reduction in interconnection traffic, the improvement

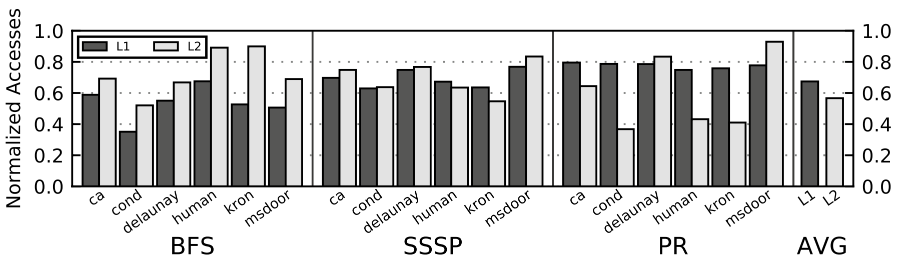


Fig. 12 Normalized accesses to L1 and L2 caches of the IRU enabled GPU system versus the Baseline GPU system (GTX 980 GPU with parameters shown in Table 2). Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset

on memory coalescing, the IRU filtering capabilities, and the overall speedup and energy savings of our proposed GPU+IRU with respect to the baseline GPU.

6.1 Memory pressure reduction

IRU's main functionality is to reorder irregular accesses improving their memory coalescing thus reducing the overall contention in the memory hierarchy. Figure 12 shows how the IRU consistently reduces accesses and contention on both L1 and L2 across all graph algorithms and datasets. Accesses to L1 and L2 are reduced to as low as 35% and 36% for the *cond* benchmark on BFS and PR, respectively. Overall, accesses are reduced to 67% and 56% for L1 and L2 caches on average.

This important reduction comes from several factors. First, the IRU reordering of irregular accesses improves coalescing reducing the accesses to L1. Second, IRU reorders requests across SMs, so it collocates accesses of a particular memory block to a single SM, avoiding data replication across L1 data caches, improving hit ratios. Third, reduced accesses to L1 avoid capacity and conflict misses, improve data thrashing and consequently reduce L2 accesses. Finally, IRU filtering further reduces accesses by removing/merging duplicated elements, that avoids additional memory accesses.

L2 accesses reduction is greater than in L1 in some benchmarks for SSSP and PR graph algorithms. Many indices reordered by the IRU on SSSP and PR are used for irregular accesses performed by atomic instructions. In GPGPU-Sim atomic operations bypass the L1 and are handled at the memory partitions. IRU coalescing and filtering improvement for these operations reduces L2 accesses but not L1 accesses, explaining the larger reduction in L2 accesses compared to L1 for SSSP and PR. Note that atomic operations within a warp are coalesced as long as different threads access different parts of a cache line.

We have also analyzed the impact of the IRU in the Network-on-Chip (NoC) that interconnects the Streaming Multiprocessors (SM) with the Memory Partitions (MP). Figure 13 shows the normalized traffic in the NoC. As it can be seen, the IRU consistently reduces interconnection traffic across all graph algorithms and datasets. Traffic between SM and MP is reduced to as low as 23% for the *human* benchmark on PR, overall reducing NoC traffic to 54% of the original interconnection traffic.

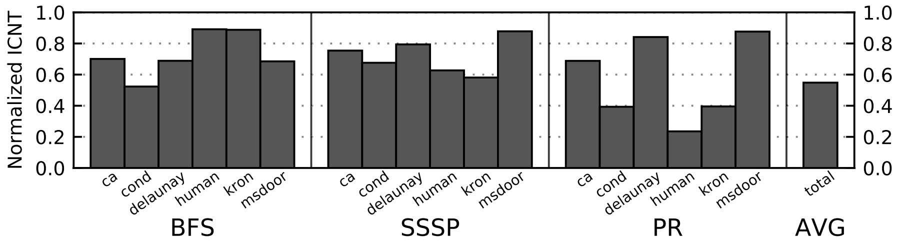


Fig. 13 Normalized interconnection traffic between SM (Streaming Multiprocessors) and MP (Memory Partitions) for the IRU enabled GPU system over the Baseline GPU system (GTX 980 GPU with parameters shown in Table 2). Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset

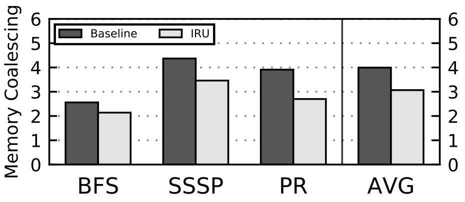


Fig. 14 Improvement in memory coalescing achieved with the IRU over the Baseline GPU system (GTX 980 GPU with parameters shown in Table 2). Vertical axis shows the number of memory requests sent to the L1 cache on average per each memory instruction

This reduction is due to several factors. First, the improved memory coalescing results in a more efficient use of the L1 data cache, significantly reducing the number of misses. Second, filtering also contributes to lower L2 accesses which reduces interconnection contention. Finally, the extended ISA instructions allow reduced traffic by issuing a single request to the IRU that receives two replies, whereas the baseline GPU would have issued two requests and two replies in order to gather data in different frontiers. Note that the IRU API allows to gather an index together with an extra attribute (e.g., weight of edge in a graph) with just one memory request as explained in Sect. 4.

Figure 14 shows the improvement in memory coalescing delivered by the IRU. A higher number indicates that more accesses are required to serve each warp memory request, with a maximum of 32 accesses per request, and a minimum of 1 access in the best scenario. The IRU improves the overall coalescing for every graph algorithm from 4 to 3 accesses per memory requests on average. This improvement is significant given that the filtering schemes that some of the algorithms employ, combined with the filtering applied by the IRU, reduce the potential for coalescing memory requests, since filtering removes some duplicated elements whose accesses could be coalesced. Nonetheless, memory coalescing is significantly improved, reducing the pressure on the memory hierarchy.

Finally, main memory accesses are reduced by 4% due to reduced L2 misses as a result of reduced accesses. Overall, reordering and filtering techniques allow the IRU to deliver significant improvements in memory coalescing and reduce contention in multiple levels of the memory hierarchy.

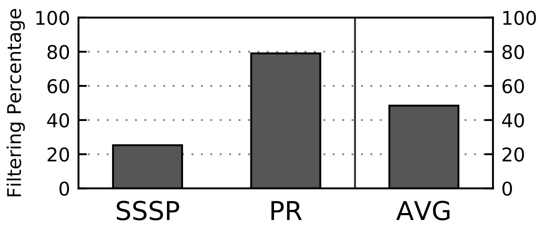


Fig. 15 Percentage of elements that are filtered out in our IRU-enabled GPU system

6.2 Filtering effectiveness

The IRU hardware provides filtering capabilities without complex additional hardware. Figure 15 shows the percentage of elements (i.e., indices with their adjacent data) processed by the IRU which are filtered out or merged. We apply the filtering to both SSSP and PR, achieving 23% and 79% workload filtering, respectively. On average, 48.5% of the elements are filtered out by the IRU. Note that this high percentage does not directly indicate that a similar amount of accesses to memory are avoided with respect to the baseline GPU, as state-of-the-art CUDA implementations of SSSP and PR include sophisticated mechanisms to filter our duplicated elements. However, in our proposed scheme the filtering is performed by the IRU hardware, whereas in the baseline GPU this filtering process is done in software. Hence, our proposal is effective at filtering/merging duplicated elements by leveraging the IRU hardware that is already available for the reordering operation, avoiding costly software filtering schemes of graph algorithms.

6.3 Performance and energy evaluation

The IRU provides performance improvements across all algorithms and benchmarks, as shown in Fig. 16. On average, the IRU achieves a speedup of 1.33x, with average speedups of 1.16x, 1.14x and 1.40x for BFS, SSSP and PR, respectively. PR exhibits higher speedups due to larger reduction in L2 accesses achieved by the filtering, which avoids costly atomic L2 accesses. SSSP achieves the lowest speedup due to lower filtering effectiveness. Overall, performance improvements come from two sources. First, the IRU improved memory coalescing by reordering of indices used for irregular accesses, which reduces contention on the memory hierarchy. Second, the IRU filtering and merging that enables further reduction in memory accesses and avoids wasted cycles in the functional units of the GPU due to processing redundant elements.

Figure 16 also shows the energy savings achieved by the IRU, which are significant across all graphs and datasets. On average, the IRU achieves an energy reduction of 13%, with reductions of 17%, 5% and 15% for BFS, SSSP and PR, respectively. Energy savings are more limited than performance improvements since the IRU greatly reduces L1 and L2 accesses but achieves a more modest reduction in main memory accesses. Note that main memory represents a very significant portion of the total energy consumed. The IRU energy overhead represents a small 0.5%

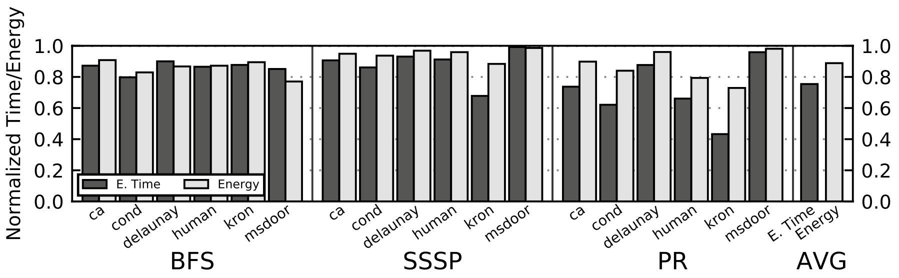


Fig. 16 Normalized execution time and energy consumption reduction in the IRU enabled GPU with respect to the baseline GPU system (GTX 980 GPU with parameters shown in Table 2). Significant speedups and energy savings achieved across BFS, SSSP and PR graph algorithms and every dataset

of the final energy. Overall, energy savings are obtained from several sources. First, the reduced accesses to L1 and L2 and reduced contention in the memory hierarchy. Second, the reduced execution time cuts down on the static power and thus, the overall energy consumption of the GPU system. Third, the energy-efficient IRU which enables the reduction in accesses and contention and allows more efficient hardware-based filtering than the costly software-based mechanisms employed by graph applications. Finally, the IRU reordering leads to a reduction in main memory accesses which further reduces energy.

6.4 Area evaluation

Our evaluation of the IRU energy and area estimations indicates that the IRU requires a total of 23.9 mm^2 when adding up all the 4 partitions of our GPU system with a GTX 980, each partition being 5.98 mm^2. The entire IRU represents 5.6% of the total GPU area. Overall, the IRU is a very compact and efficient hardware which manages to deliver significant performance and energy savings with small area requirements.

7 Related work

Irregular programs on GPGPU architectures face many challenges resulting in low GPU utilization and poor performance. Several previous works have thoroughly analyzed the causes of these inefficiencies, that boil down to control flow divergence and memory accesses irregularity [11, 12, 20, 21]. Nonetheless, if these issues are overcome, irregular applications can greatly benefit of the high parallelism that GPU architectures offer. Over the recent years, several works have approached the topic of efficient and improved irregular programs on GPGPU architectures.

Some solutions approach the branch divergence issue by providing load balancing solutions [15, 30] to improve utilization of execution units. Others provide thread remapping over warps to improve branch divergence [31]. Some memory divergence approaches propose modifying software data structures [9, 16, 32], whereas others such as HALO [33] provide static reordering of a graph to improve data locality. Many specialized works have focused on GPU execution of irregular Sparse Matrix Vector Multiplication (SpMV) and Matrix Matrix Multiplication (GEMM) by proposing software approaches that reorder the matrices dataset [34], and algorithms tailored for specific matrix data characteristics [35], and row reordering techniques [36] to improve data locality among processed rows. Finally, other works prove the NP-completeness of finding the data layouts through data repositioning that minimize uncoalesced memory accesses and propose software algorithms to attain them [37]. These works propose methods requiring significant programming effort, as they require changing algorithms and data structures or profound hardware knowledge. In contrast, our IRU solution requires very lightweight changes of the algorithms and does not require profound knowledge of the inner working of the GPU memory hierarchy to improve memory coalescing and resolve contention issues.

Other approaches explore microarchitectural improvements transparent to the programmer, or with some minor involvement to achieve the desired result. Extensive research has been done on flexible cache solutions [38–40] that adapt for fine-grained and coarse-grained accesses while other works resort to cache bypassing mechanisms [41]. Works such as LAMAR [42] explore sizable GPU architecture and memory hierarchy modifications to detect and provide fine and coarse grained accesses throughout the memory system. Other works propose hybrid software and hardware approaches that enable data-dependent aware dynamic scheduling [43] or provide prefetching of irregular accesses [44] to registers to avoid early data eviction. Finally, works such as D2MA [45] and Stash [46] set to provide mechanisms to manage global data allocation to shared memory, with the objective to increase capacity close to the cores and improve memory hierarchy and overall performance. The aforementioned works leverage hardware solutions that work around or ameliorate the consequences of low memory coalescing by providing mechanisms to lower memory contention. In contrast, our IRU provides tools to amend the cause, not the consequence, of the high memory contention, i.e., the poor memory coalescing. Intermediate approaches have explored extending the GPU architecture with custom purpose hardware units. SCU [8] proposes a programmable GPU hardware extension for graph processing that is tailored to stream compaction operations required for graph processing. Meanwhile, the GPU is employed to execute the graph processing workload part that is most well suited for, achieving significant performance improvements. In comparison, the IRU is a more flexible extension, with a more generic and reusable API tailored to general irregular accesses patterns. Furthermore, the SCU requires significant changes in the application, since entire kernels are replaced by calls to the SCU, whereas other kernels must be adapted. Our solution requires minor changes to the application as described in Sect. 4.

Finally, many works propose to replace entirely the GPU with special purpose accelerators custom-made for graph processing, which set aside the GPU due to fundamental limitations and exploit deep knowledge of graphs data structures. Proposals include standalone approaches such as TuNao [47], Dram-based Graphicionado [48] or PIM-based GraphH [49]. In contrast, our IRU solution leverages the popularity of GPU architectures and provides generic solutions that bring the performance and efficiency of GPU architectures for low performing irregular programs.

8 Conclusions

In this paper, we propose the Irregular accesses Reorder Unit (IRU), a GPU extension that improves performance and energy efficiency of irregular applications. Efficient execution of irregular applications on GPU architectures is challenging due to low utilization and poor memory coalescing, which force programmers to carry out complex code optimization techniques to achieve high performance. The IRU is a novel hardware unit that delivers improved performance of irregular applications by reordering data serviced to threads. This reordering is enabled by relaxing the strict relationship between threads and data processed. We further extend the IRU to filter out and merge repeated elements while performing the reordering, this results in increased performance by largely reducing redundant GPU workload. The IRU reordering and filtering schemes deliver 1.32x improved memory coalescing, while reducing the traffic in the memory hierarchy by 46%. Our IRU augmented GPU system achieves on average 1.33x speedup and 13% energy savings for a diverse set of graph-based applications and datasets, while incurring in a small 5.6% area overhead.

Acknowledgements This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00 and the ICREA Academia program.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability The datasets generated during the current study are available from the corresponding author on reasonable request.

References

1. Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation
2. Li J, Ranka S, Sahni S (2011) Strassen's matrix multiplication on gpus. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp 157–164. IEEE
3. Root C, Mostak T (2016) MAPD: a GPU-powered big data analytics and visualization platform. In: ACM SIGGRAPH 2016 Talks, pp 1–2
4. Yan M, Chen Z, Deng L, Ye X, Zhang Z, Fan D, Xie Y (2020) Characterizing and understanding GCNs on GPU. IEEE Comput Archit Lett 19(1):22–25
5. Chong J, Gonina E, Keutzer K (2011) Efficient automatic speech recognition on the GPU. In: GPU Computing Gems Emerald Edition, pp 601–618

6. Krizhevsky A, Sutskever I, Hinton G.E (2012) Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp 1097–1105
7. Kato S, Tokunaga S, Maruyama Y, Maeda S, Hirabayashi M, Kitsukawa Y, Monrroy A, Ando T, Fujii Y, Azumi T (2018) Autoware on board: enabling autonomous vehicles with embedded systems. In: 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), pp 287–296 . IEEE
8. Segura A, Arnau J.-M, González A (2019) SCU: a GPU stream compaction unit for graph processing. In: Proceedings of the 46th International Symposium on Computer Architecture, pp 424–435
9. Nodehi Sabet AH, Qiu J, Zhao Z (2018) TIGR: Transforming irregular graphs for GPU-friendly graph processing. ACM SIGPLAN Notices 53(2):622–636
10. Beamer S (2016) Understanding and improving graph algorithm performance. Ph.D. thesis, UC Berkeley
11. Burtscher M, Nasre R, Pingali K (2012) A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp 141–151. IEEE
12. O'Neil MA, Burtscher M (2014) Microarchitectural performance characterization of irregular GPU kernels. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp 130–139 . IEEE
13. Sengupta S, Harris M, Garland M (2008) Efficient parallel scan algorithms for GPUs. NVIDIA, Santa Clara,CA, Tech. Rep. NVR-2008-003 1(1):1–17
14. Billeter M, Olsson O, Assarsson U (2009) Efficient stream compaction on wide SIMD many-core architectures. In: Proceedings of the Conference on High Performance Graphics 2009, pp 159–166
15. Merrill D, Garland M, Grimshaw A (2015) High-performance and scalable GPU graph traversal. ACM Trans Parallel Comput (TOPC) 1(2):1–30
16. Wang Y, Davidson A, Pan Y, Wu Y, Riffel A, Owens JD (2016) Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp 1–12
17. Low Y, Gonzalez JE, Kyrola A, Bickson D, Guestrin CE, Hellerstein J (2014) Graphlab: a new framework for parallel machine learning. arXiv preprint arXiv:1408.2041
18. Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2008) The graph neural network model. IEEE Trans Neural Netw 20(1):61–80
19. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65
20. Lumsdaine A, Gregor D, Hendrickson B, Berry J (2007) Challenges in parallel graph processing. Parallel Process Lett 17(01):5–20
21. Xu Q, Jeon H, Annavaram M (2014) Graph processing on GPUs: Where are the bottlenecks? In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp 140–149 . IEEE
22. Davidson A, Baxter S, Garland M, Owens JD(2014) Work-efficient parallel GPU methods for single-source shortest paths. In: 2014 IEEE 28th International Conference on Parallel and Distributed Processing Symposium, pp 349–359. IEEE
23. Geil A, Wang Y, Owens JD (2014) WTF, GPU! computing Twitter's who-to-follow on the GPU. In: Proceedings of the Second ACM Conference on Online Social Networks, pp 63–68. ACM
24. Bell N, Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p 18. ACM
25. Bakhoda A, Yuan GL, Fung W.W, Wong H, Aamodt TM (2009) Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium On Performance Analysis of Systems and Software, 2009. ISPASS 2009, pp 163–174. IEEE
26. Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2009) MCPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 42nd Annual IEEE/ACM International Symposium On Microarchitecture, 2009. MICRO-42, pp 469–480. IEEE
27. Leng J, Hetherington T, ElTantawy A, Gilani S, Kim NS, Aamodt TM, Reddi VJ (2013) Gpuwattch: enabling energy optimizations in gpgpus. In: ACM SIGARCH Computer Architecture News, vol. 41, pp 487–498. ACM
28. Davis TA, Hu Y (2011) The university of florida sparse matrix collection. ACM Trans Math Softw (TOMS) 38(1):1

29. DIMACS: 10th DIMACS Implementation Challenge—Graph Partitioning and Graph Clustering (2010). https://www.cc.gatech.edu/dimacs10/
30. Khorasani F, Gupta R, Bhuyan LN (2015) Scalable SIMD-efficient graph processing on GPUs. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp 39–50. IEEE
31. Fung WW, Sham I, Yuan G, Aamodt TM (2007) Dynamic warp formation and scheduling for efficient GPU control flow. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pp 407–420. IEEE
32. Gharaibeh A, Reza T, Santos-Neto E, Costa LB, Sallinen S, Ripeanu M (2013) Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv preprint arXiv:1312.3018
33. Gera P, Kim H, Sao P, Kim H, Bader D (2020) Traversing large graphs on GPUs with unified memory. Proc VLDB Endow 13(7):1119–1133
34. Pichel JC, Rivera FF, Fernández M, Rodríguez A (2012) Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. Microprocess Microsyst 36(2):65–77
35. Rivera C, Chen J, Xiong N, Song SL, Tao D (2020) Ism2: optimizing irregular-shaped matrix-matrix multiplication on GPUs. arXiv preprint arXiv:2002.03258
36. Jiang P, Hong C, Agrawal G (2020) A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp 376–388
37. 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 Notices 48(8):57–68
38. Kumar S, Zhao H, Shriraman A, Matthews E, Dwarkadas S, Shannon L (2012) Amoeba-cache: adaptive blocks for eliminating waste in the memory hierarchy. In: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp 376–388. IEEE
39. Li B, Sun J, Annavaram M, Kim NS (2017) Elastic-cache: GPU cache architecture for efficient fine- and coarse-grained cache-line management. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 82–91. IEEE
40. Guo H, Huang L, Lü Y, Ma S, Wang Z (2018) Dycache: dynamic multi-grain cache management for irregular memory accesses on GPU. IEEE Access 6:38881–38891
41. Chen X, Chang L-W, Rodrigues CI, Lv J, Wang Z, Hwu W-M (2014) Adaptive cache management for energy-efficient GPU computing. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp 343–355. IEEE
42. Rhu M, Sullivan M, Leng J, Erez M (2013) A locality-aware memory hierarchy for energy-efficient GPU architectures. In: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 86–98. IEEE
43. Yan M, Hu X, Li S, Basak A, Li H, Ma X, Akgun I, Feng Y, Gu P, Deng L, Xiaochun Y, Zhimin Z, Dongrui F, Yuan X (2019) Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp 615–628
44. Lakshminarayana NB, Kim H (2014) Spare register aware prefetching for graph algorithms on GPUs. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp 614–625 . IEEE
45. Jamshidi DA, Samadi M, Mahlke S (2014) D2ma: accelerating coarse-grained data transfer for GPUs. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp 431–442
46. Komuravelli R, Sinclair MD, Alsop J, Huzaifa M, Kotsifakou M, Srivastava P, Adve SV, Adve VS (2015) Stash: have your scratchpad and cache it too. ACM SIGARCH Comput Arch News 43(3S):707–719
47. Zhou J, Liu S, Guo Q, Zhou X, Zhi T, Liu D, Wang C, Zhou X, Chen Y, Chen T (2017) TUNAO: a high-performance and energy-efficient reconfigurable accelerator for graph processing. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 731–734. IEEE
48. Ham T.J, Wu L, Sundaram N, Satish N, Martonosi M (2016) Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 1–13. IEEE
49. Dai G, Huang T, Chi Y, Zhao J, Sun G, Liu Y, Wang Y, Xie Y, Yang H (2018) Graphh: a processing-in-memory architecture for large-scale graph processing. IEEE Trans Comput Aided Des Integr Circuits Syst 38(4):640–653

Authors and Affiliations

Albert Segura[1] · **Jose Maria Arnau**[1] · **Antonio Gonzalez**[1]

Jose Maria Arnau
jarnau@ac.upc.edu

Antonio Gonzalez
antonio@ac.upc.edu

[1] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya (UPC), Campus Nord, Jordi Girona 1-3, 08034 Barcelona, Spain