

Irregular Accesses Reorder Unit: Improving GPGPU Memory Coalescing for Graph-Based Workloads

Albert Segura, Jose-Maria Arnau, Antonio González
 Department of Computer Architecture
 Universitat Politècnica de Catalunya - BarcelonaTech (UPC)
 asegura, jarnau, antonio@ac.upc.edu

ABSTRACT

GPGPU architectures have become established as the dominant parallelization and performance platform achieving exceptional popularization and empowering domains such as regular algebra, machine learning, image detection and self-driving cars. However, irregular applications struggle to fully realize GPGPU performance as a result of control flow divergence and memory divergence due to irregular memory access patterns.

To ameliorate these issues, programmers are obligated to carefully consider architecture features and devote significant efforts to modify the algorithms with complex optimization techniques, which shift programmers priorities yet struggle to quell the shortcomings. We show that in graph-based GPGPU irregular applications these inefficiencies prevail, yet we find that it is possible to relax the strict relationship between thread and data processed to empower new optimizations.

Based on this key idea, 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 which significantly improves memory coalescing, and allows increased performance and energy efficiency. Additionally, the IRU is capable of filtering and merging duplicated irregular access which further improves graph-based irregular applications. Programmers can easily utilize the IRU with a simple API, or compiler optimized generated code with the extended ISA instructions provided.

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 and 13% improvement in performance and energy savings respectively, while incurring in a small 5.6% area overhead.

1. INTRODUCTION

Since its popularization over the last decade, GPGPU architectures have enabled a broad domain of new applications by boosting regular algebra computations [3, 26], empowering Big Data analytics [36] and deploying Machine Learning [42] in numerous fields such as speech recognition [8], image detection [22] and self-driving cars [19]. GPGPU architectures excel at processing highly-parallelizable throughput oriented

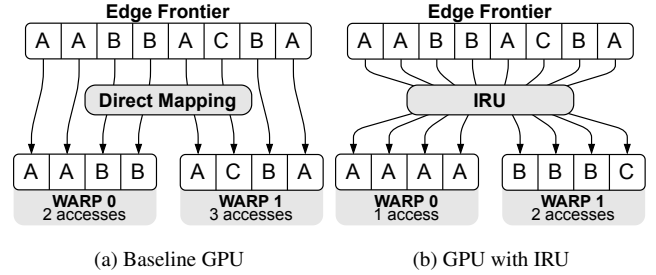


Figure 1: Memory Coalescing improvement achieved by employing the IRU (1b) to reorder data elements that generate irregular accesses versus a Baseline GPU (1a) execution.

applications, which exhibit regular execution and memory access patterns. Applications well suited for GPGPU architectures closely match these characteristics but others exhibiting sparse or irregular memory accesses or branch divergence, can show seriously hindered GPGPU efficiency [38]. Graph processing algorithms can potentially benefit from the highly-parallel GPGPU architectures. However, they process unstructured and irregular data, which results in sparse and unpredictable memory access patterns [32]. In addition, graph processing shows extremely low computation to memory access ratio [2], which further hinders the GPGPU efficiency.

To mitigate the aforementioned problems, GPGPU algorithms have to carefully consider the underlying hardware and adapt the algorithm to minimize branch divergence and improve memory coalescing (i.e. collocated accesses to memory), among other performance optimizations [6, 33]. Graph algorithms employ many such techniques, such as scan algorithms [39] which are leveraged for data compaction [5] that gathers data to be accessed sparsely into a compacted data array improving data locality and memory coalescing. 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.

GPGPU programming models such as CUDA employ threads to exploit parallelism, each 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 Figure 1a. For a regular program, this assignment is highly effective at achieving good utilization of resources without inefficiencies (e.g. vector addition, where each thread in a warp processes consecutive data in memory achieving regular behavior). For programs that exhibit irregular memory accesses this simple assignment might cause a degradation in utilization, as the GPU is unable to achieve good memory coalescing in a warp, and can result in poor data locality (e.g. graph processing algorithms, where each thread processes a given node of the graph and has to fetch its adjacent ones). To ameliorate these issues, a more complex thread to data assignment or additional preprocessing (e.g. reordering) can be performed by the programmer, but at the cost of additional algorithm complexity and computational cost.

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 bad memory coalescing, producing intra-warp memory divergence and reducing GPU efficiency significantly. Second, these issues are, in the best of cases, 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 of the application. 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 the GPGPU architecture 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, very tightly integrated hardware in the GPU which is used through a set of new ISA instructions which ultimately 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 greatly improve memory coalescing. Figure 1 shows conceptually how the IRU assigns data to the threads and achieves an improvement on 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 data),

which reduces useless resource utilization of the GPU 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 which enables improved performance of sparse and irregular accesses by reordering data serviced to each thread. We further extend the IRU to filter repeated elements in graph-based applications, largely reducing GPU redundant workload.
- We propose an ISA extension and high-level API and show how modern graph-based applications can easily leverage the IRU hardware.
- Overall the GPU architecture with our IRU improves memory coalescing by a factor of 1.32x and reduces the overall memory hierarchy traffic by 46%, resulting in 1.33x and 13% speedup and energy savings respectively 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 applications, in particular graph processing, on GPGPU architectures. Section 3 presents the architecture of the IRU, whereas Section 4 describes its API and usage for graph applications. Section 5 describes the evaluation methodology and Section 6 provides the experimental results. Section 7 reviews relevant related work and, finally, Section 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. Many applications fit these characteristics which allow efficient utilization of the GPU resources and high performance. GPU's high IPC is enabled by its Single-Instruction, Multiple-Threads (SIMT) pipeline, which leverages the advantage of decoding a single instruction used by multiple threads, each operating on different data. The threads in a warp execute in a lock-step manner, and so to completely utilize the Execution Units (EU) it is necessary non-diverging, regular applications. Furthermore, to sustain high IPC, significant memory bandwidth is required which is accomplished with high Memory-Level Parallelism (MLP) by leveraging warp-level coalescing and concurrent execution of many threads, achieving a trade-off between increased memory bandwidth at the expenses of increased latency. Regular applications experience regular, predictable memory access patterns that maximize memory coalescing and MLP.

On the other hand, many applications that show a more irregular behavior with unpredictable irregular memory access patterns can also benefit from processing many independent data in parallel. Yet, for these applications GPGPU architectures are unable to provide enough memory bandwidth due to a huge portion of the threads generating uncoalesced accesses. Performance is additionally hampered when all EUs are not utilized due to divergent thread execution. A memory instruction producing irregular accesses achieves poor memory coalescing; since the accesses generated by the threads in a single warp will likely not be collocated into the same memory block, an individual memory request will have to be issued for every uncoalesced thread, one per thread in the worse case. In contrast, for a regular application, every access generated by a thread in a warp will potentially be coalesced, issuing a single memory request to the L1 cache.

Consequently, irregular application overhead is very significant, potentially up to 32x more memory requests (assuming a typical warp size of 32 threads), which increases both utilization of the LD/ST unit and instruction latency and puts higher pressure on the L1 and the whole memory hierarchy. In addition, every warp instruction requires more resources to handle misses on L1, such as miss status holding registers (MSHRs) and entries in the miss queue, a problem aggravated by GPUs L1 small capacity ratio of cache lines per thread compared to CPUs. All these factors combined increase significantly the contention on the L1 and its miss ratio due to conflict and capacity misses. Finally, the interconnection traffic congestion increases, L2 observes similar problems to the ones described for L1, and main memory accesses increase as a consequence of increased L2 misses.

To reduce the overhead caused by irregular accesses, significant changes have to be applied to the algorithm and its data structures as to utilize more efficiently the GPU resources. Generic optimization approaches include the use of the shared memory present in the Streaming Multiprocessors (SM) of the GPU, which provide reduced latency and allow banked accesses of uncoalesced accesses; while other approaches favor merging kernels to reuse memory requests while increasing use of registers and contention on registers files. Graph algorithms use techniques such as data compaction using scan algorithms, which reduce sparse accesses and improve locality by gathering sparse data in a compacted data array; as well as load balancing techniques employed to leverage threads that collaborate to reduce branch divergence and improve memory coalescing.

Overall, irregular applications can and do benefit from the high performance delivered by the huge parallelism of GPU architectures, but the architecture has many pitfalls when it comes to enabling high performance and utilization of irregular algorithms. Significant programmer effort, code complexity and underlying hardware knowledge is 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 [28, 37] and Data Analytics [44] are modeled using graphs which represent relationships between the elements on a set of data. GPGPU architectures enable fast parallel exploration and processing

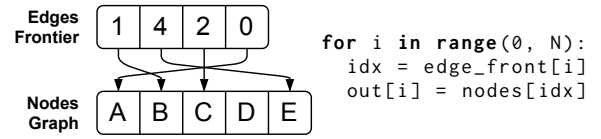


Figure 2: The graph edges frontier produces irregular accesses when accessing the nodes data in the graph.

of the nodes and connections (i.e. edges) of a graph. Nonetheless, graph exploration is low-computation intensive [2], unstructured and irregular [29, 41] with sparse, irregular and highly unpredictable access patterns due to the unpredictable and irregular nature of the relationships expressed in a graph, so proper GPGPU resource utilization is hard to achieve.

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 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 on the nodes array, which is an intrinsic part of graph exploration algorithms and a cause of the previously mentioned inefficiencies. In this work we focus on common graph algorithms, in particular push implementations of Breadth-First Search (BFS) [31], Single-Source Shortest Paths (SSSP) [10] and PageRank (PR) [14].

GPGPU graph processing adopts many strategies to ameliorate these inefficiencies. First, data structures that efficiently represent the graph data in a compact manner using the Compressed Sparse Row (CSR) [4] format. Second, to cut down on sparse accesses, stream compaction algorithms [5] are used to gather data in contiguous memory improving data locality and coalescing. Finally, the connectivity in a dataset is very irregular and unpredictable; some nodes might have a very significant number of adjacent edges while others will have few or none. This disparity leads to unbalanced processing where some nodes require more processing, which results in poor EU utilization and leads to low IPC. To counter this issue, load balancing techniques [31] are used to leverage the threads in warps and thread blocks to cooperatively process data from the more processing demanding nodes.

Overall, while many techniques enable efficient GPGPU graph processing execution, significant changes are required to implement these optimizations and reduce the GPGPU architecture shortcomings for irregular applications. Modern graph applications experience significantly bad memory coalescing of 3.9 accesses per warp, and utilization is consequently low reaching 13.5%. These numbers point out the importance of improving the hardware support for irregular applications and graph algorithms in GPGPU architectures.

3. IRREGULAR ACCESSSES 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.

We propose to extend the GPGPU with the IRU to reduce the overheads caused by irregular accesses. The IRU is a com-

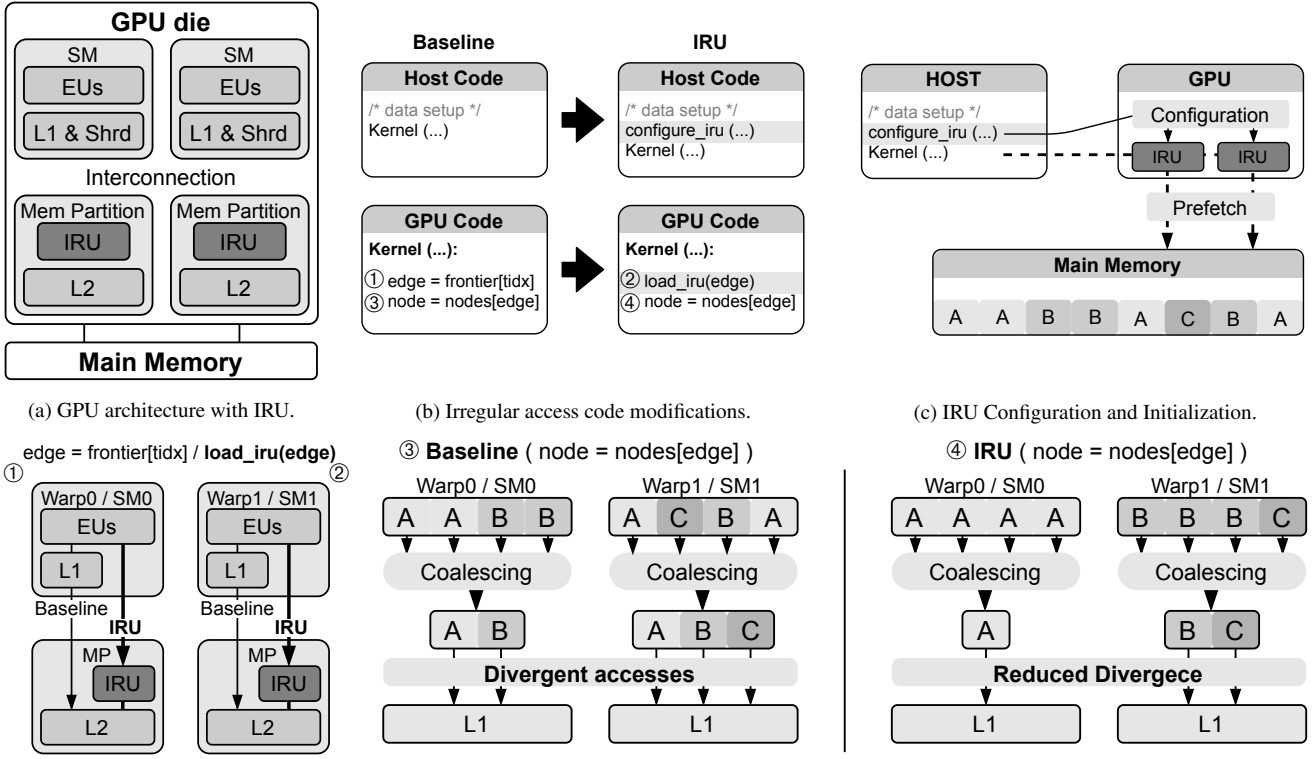


Figure 3: IRU integration with the GPU at different levels: architectural (a), program model (b) and execution (c,d,e). The execution showcases how the program (b) works on the Baseline and the IRU operate with two warps and data from Figure 1.

compact and efficient hardware unit integrated into the Memory Partition (MP) of the GPU architecture as seen in Figure 3a, which incurs in very small energy and area overheads. The IRU leverages the observation that GPU programs employ threads as a mean to convey parallelism; they are in many occasions independent of the data that they process. The main goal of the IRU is to process the indices used to perform irregular accesses, reorder and redistribute them. The reordering aggregates indices that access the same memory block and services them to a requesting warp, improving the collocation of irregular accesses and thus increasing memory coalescing. In turn, the improved memory coalescing, reduces congestion of the resources of the LD/ST unit, L1, interconnection, L2 and main memory is significantly reduced. In addition, the reordering is performed across all the indices accessed by all the SM, and so, 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 4 shows the average normalized execution of a warp of 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. The overhead incurred by the IRU servicing the load is more than offset by the additional performance gained from the reduction of the overheads due to improved memory coalescing of the targeted irregular access.

The IRU processes the indices of a target irregular instruction, with the objective to optimize its coalescing. Additionally, the elements processed contain more data than just the indices, as mandated by the API described in Section 4.

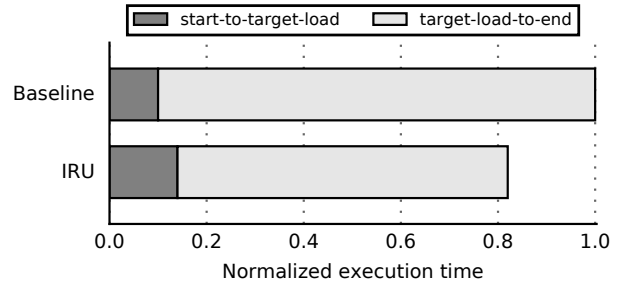


Figure 4: Warp average normalized execution with and without IRU. The dark bar indicates execution time until the target load is serviced, and the light bar from service to finalization. The IRU achieves speedups despite the overhead introduced.

While, these data are not used for the IRU coalescing logic, since the indices remain the information that the IRU utilizes to improve memory coalescing, it is responsible to fetch, generate and reply to the SM the additional data.

3.1 GPU Integration

The IRU integration into the GPU is covered in Figure 3, showing architectural 3a, programming 3b and execution 3c - 3e integration. The execution shows how the Baseline and the IRU modified GPU programs in Figure 3b, operate with the two warps and data from Figure 1.

The Baseline program performs a regular access ① 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 a simple modification explored in Section 4. The baseline code is executed by the GPU as follows. First, the two warps

retrieve the indices performing regular accesses to the L1, as seen in Figure 3d. Afterwards, Figure 3e shows how they perform irregular accesses to the L1 with the retrieved indices which, due to the high divergence, result in many accesses ③.

In contrast, the IRU program first introduces a configuration step performed on the host, shown in Figure 3c, 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 access. Further IRU capabilities are enabled and used with optional parameters on overloaded functions, reviewed in Section 4. Afterwards, when the kernel execution starts, the IRU triggers the prefetching of the indices from L2 and memory, which are then autonomously reordered in the IRU hash. The IRU activity is overlapped with the execution of the kernel, and disabled when all the data is processed or if not used for the kernel in execution.

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 Figure 3d. The IRU replies with reordered indices either instantly, if they are ready, or otherwise after a timeout to avoid starvation. 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 Figure 3e.

3.2 Hardware Overview and Processing

The internal pipelined hardware of the IRU is shown in Figure 5a. It is composed of a number of blocks each with specific purpose, simple logic and buffering of data. The main purpose of the IRU, which is to reorder indices to improve memory coalescing, is accomplished with the use of a hash located inside the *Reordering Hash* block. Instead of multiple private hashes, there is a single logical hash 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. We have observed that the degree of memory coalescing is significantly affected if each IRU hash 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 executions. Alternatively, requests can be configured to bypass L2, which could become beneficial for streaming kernels that do not reuse the data.

The overall internal processing of the IRU is described in Figure 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 first initialized from the Host by executing the *configure_iru* function with the corresponding data ①. The *Prefetcher* later uses this information to determine the addresses to prefetch when the GPU kernel starts executing ②, then it begins issuing a limited number of on-

the-fly prefetches to avoid saturating memory bandwidth and degrading performance. Each IRU only prefetches information from its corresponding memory partition, in Figure 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 is stored in a FIFO queue to be later processed.

Afterwards, the *Classifier* block processes the prefetched data ③ by splitting it into several smaller FIFO queues, an element per cycle per queue. The smaller FIFO queues contain the elements that will be inserted in the hash or forwarded through the interconnection. A hashing function of the element is used to determine which hash entry it is mapped to and, therefore, if it will access a local bank or must be sent through the interconnection. Finally, the *Data Processing* block retrieves elements from both the smaller FIFO queues and the ring, prioritizing the latter, and forwards it to the ring or inserts it to the local hash ④. On Figure 5e, the elements labeled A are inserted into the local hash, 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* ⑤. 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 is 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, 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 Figure 5d, the elements labeled B are determined to correspond to another IRU partition and so are inserted in the ring ⑥. Meanwhile, data from the neighbor partition is received (indices A and C are determined to correspond to IRU 0) ⑦.

Lastly, the elements corresponding to this IRU partition are gathered from the ring and inserted into the reordering hash. When the *Data Replier* detects a hash entry that is complete, or enough data is available to reply a request, the oldest request is replied back to the SMs with that entry reordered elements ⑧, and the data is evicted from the hash. The data used for the reply (four A) are the indices used for the irregular access being optimized, but additionally more data might be processed per element, in which case multiple replies would be issued, at most two additional replies.

Additionally, a timeout is employed to avoid excessively delaying a request. Once the timeout is reached, it then fetches data from the hash with the best coalesced data entry present, and replies once enough data is retrieved, effectively trading-off worse coalescing for lower latency. Additionally, 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 IRU in the memory partitions, and requests can be replied by IRU partitions other than the original. Finally, when no more data is left to be inserted into the IRU, the *Data Replier* replies to the SM by merging the remaining hash entries which might not be full. These entries selection

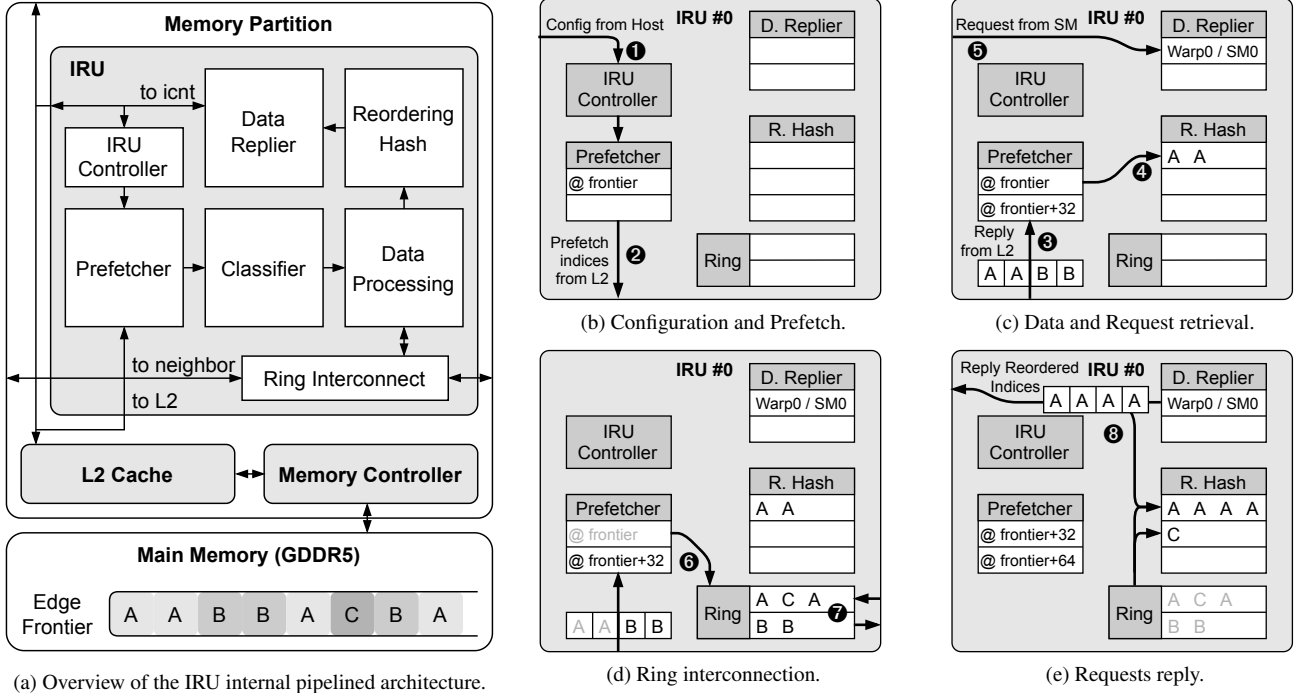


Figure 5: Architecture and the internal processing performed by the IRU. The indices in memory (from Figure 1) are processed by two IRU partitions (IRU 0 shown), which is later replied to a request coming from Warp 0 in SM 0.

avoids splitting a hash entry between two replies, which would consequently impact memory coalescing.

3.3 Reordering Hash

The *Reordering Hash* drives the IRU, it 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. Furthermore, the hash function key that points to an entry is generated from the value being inserted into the hash entry. The computation of the hash function collocates in a single hash 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, an insertion allows to insert 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 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 which avoids handling conflicts that would have the conflicting elements gathered and replied back to the SM without achieving any improved coalescing. Furthermore, a good dispersion hash function and properly sized hash tables limits the amount of conflicts and contains the negative impact 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 impairing memory coalescing as severely.

Some API operations described in Section 4 require ad-

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

4. IRU PROGRAMMABILITY

Ease of programability is a highly important aspect when it comes to writing efficient parallel programs, reason for which toolkits such as CUDA are very successful. As explored in Section 2, efficient irregular programs require complex optimization techniques. The IRU has been designed to be easily integrated and programmable. 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 and changes to the LD/ST unit to route these requests to the IRU. To avoid directly using ISA instructions we provide a simple API easily integrated in CUDA kernels. Furthermore, since the changes to the code are minimal, a compiler that supports the ISA extensions can generate the instrumented code, freeing the programmer from performing the optimization effort and delivering a more efficient GPGPU architecture for irregular applications.

IRU's main optimization is the reordering of indices fetched from memory that are used for irregular accesses. This optimization is based on the premise that the data assigned to the threads is independent of what thread is processing it. Consequently, to be able to correctly utilize the IRU for this optimization, the programmer has to guarantee that the reordering can be applied correctly as other data or accesses might have to be done with the new order achieved. The API provides additional functionality to facilitate this guarantee.

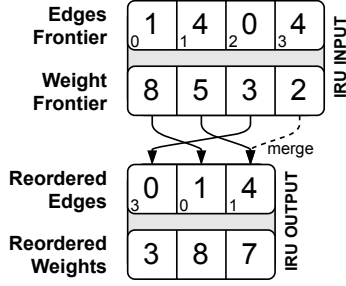


Figure 6: IRU processing of two arrays with filtering enabled. The edges is the indexing array, while the weight is the secondary array. The filtering operation is an addition.

The baseline functionality provided by the API and IRU hardware supports reorder 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 6 shows how the input data, first two rows, is 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 this case, the reordering operation can return in which position in the original array the reordered element was located. This is indicated in Figure 6 by the sub-index of the edge frontier, showing the position in the original array. This position value can be used to fetch any additional attributes required.

Graph-based algorithms process several nodes and edges simultaneously. For this reason, it is common that several edges lead to the same destination node which causes redundant work. This additional work is usually benign as the program implements filtering techniques, which are effective yet computationally costly due to synchronization requirements. To aid the program with 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 elements. The operations supported by the IRU are integer comparison and floating point addition. Figure 6 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 data, and so we extend the API to indicate if a given thread data has been filtered out. IRU groups the disabled threads in warps rather than preparing replies to warps with reordered data and disabled threads, this approach allows to minimize branch divergence, remove redundant work and improve IPC.

The API seen in Figure 7 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 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.

```

1 void configure_iru (
2     addr_t target_array,
3     size_t target_array_data_type_size,
4     addr_t indices_array,
5     addr_t secondary_array,
6     size_t number_elements,
7     filter_op_t filter_op );
8
9 __device__ bool load_iru (
10    addr_t &indices_array,
11    addr_t &secondary_array,
12    uint32_t &position );

```

Figure 7: API overloaded function declarations.

4.1 IRU enabled Graph Applications

All the previously described functionalities enable the instrumentation of state-of-the-art implementation of 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 and minimal code changes while providing efficient memory coalescing improvements. The following examples show how the *load_iru* can be used from within GPGPU kernels easily replacing existing code.

```

1 __global__ void BFS_Contract (...) {
2     int pos = blockDim.x * blockIdx.x +
3             threadIdx.x;
4     if (pos < number_elements) {
5         int edge;
6
7         #ifdef NOT_INSTRUMENTED
8             edge = edge_frontier[pos];
9         #elif USE_IRU
10            load_iru(edge);
11        #endif
12
13        // more computation ...
14        label[edge] = distance;
15    }
16 }

```

Figure 8: Simple instrumentation of a BFS algorithm Kernel using the API of the IRU.

The basic functionality of the IRU is a good fit for the BFS algorithm as illustrated in Figure 8. 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 improves.

The SSSP algorithm processes additional data per element; each edge has an associated weight value. Figure 9 shows

```

1 __global__ void SSSP_Compaction (...) {
2     int pos = blockDim.x * blockIdx.x +
3         threadIdx.x;
4     if (pos < number_elements) {
5         int edge, weight;
6
7 #ifdef NOT_INSTRUMENTED
8         edge = edge_frontier[pos];
9         weight = weight_frontier[pos];
10 #elif USE_IRU
11         load_iru(edge, weight, pos);
12 #endif
13
14         int previous =
15             atomicMin(&label[edge], weight);
16         if (previous > weight)
17             lookup[edge] = pos;
18     }
19 }

```

Figure 9: Simple instrumentation of a SSSP algorithm Kernel using the API of the IRU.

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.

```

1 __global__ void PR_Contract (...) {
2     int pos = blockDim.x * blockIdx.x +
3         threadIdx.x;
4     if (pos < number_elements) {
5         int edge;
6         float weight;
7         bool active_thread = true;
8
9 #ifdef NOT_INSTRUMENTED
10         edge = edge_frontier[pos];
11         weight = weight_frontier[pos];
12 #elif USE_IRU
13         active_thread = load_iru(edge, weight);
14 #endif
15
16         if (active_thread)
17             atomicAdd(&label[edge], weight);
18     }
19 }

```

Figure 10: Simple instrumentation of a PageRank algorithm Kernel using the API of the IRU.

Finally, the PageRank kernel shown in Figure 10 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 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 [1]. To properly integrate the IRU into the GPGPU simulator we modified the decoding to add our new instructions to the ISA, as well as incorporating small modifications to the LD/ST unit to handle the new instructions.

Table 1: IRU hardware requirements per partition.

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

Each partition of the IRU uses a 2 KB FIFO to buffer warp requests, 1.7 KB buffering of prefetching data for 8 on-the-fly simultaneous prefetches. A buffer of 1.2 KB is used internally in the Classifier block to determine the data destination. The ring requires a total of 2.8 KB space for buffering. The main component of the IRU is the hash, which is a direct mapping hash table with 1024 sets, split in 4 physical partitions. Each IRU partition is 2-way banked and holds 256 sets which amount to a total of 80 KB, 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 logic or execution unit we model area and energy consumption using CACTI [27] with a node technology of 32 nm.

GPGPU performance is modeled with GPGPU-Sim [1] and energy consumption and area with GPUWattch [24], both simulators are configured with the parameters shown in Table 2 to model our target GPU, an NVIDIA GTX 980.

Table 2: GPGPU-Sim High-Performance GTX980.

Characteristic	Configuration
GPU, Frequency	NVIDIA GTX 980, 1.27GHz
Streaming Multiproc.	16 (2048 threads), Maxwell
L1, L2 caches	32 KB, 2 MB. 128 B lines
L1, L2 MSHRs	32/32 assoc, 8/4-merge.
Memory Partitions	4 (4 channel GDDR5)
Main Memory	4 GB GDDR5, 224 GB/s

To evaluate our proposal we use state-of-the-art GPGPU implementations of BFS [31], SSSP [10], and PageRank [14] graph algorithms evaluated with benchmarks datasets, shown in Table 3, collected from well-known repositories of research graph datasets [11, 12]. These graphs are representative of different application domains with varied sizes, characteristics and degrees of connectivity.

6. EXPERIMENTAL RESULTS

In this section, we analyze how the memory hierarchy contention is reduced, the reduction of interconnection traffic, the improvement on memory coalescing, the IRU filtering capabilities, and the overall performance and energy improvement of our proposed GPU system with the IRU with respect to the baseline GPU. Our workloads are the graph algorithms BFS, SSSP and PR, that are run for a set of diverse graphs shown in Table 3.

Table 3: Benchmark graph datasets.

Graph Name	Description	Nodes (10^3)	Edges (10^6)	Avg. Degree
ca [11]	California road network	710	3.48	9.8
cond [11]	Collaboration network, arxiv.org	40	0.35	17.4
delaunay [12]	Delaunay triangulation	524	3.4	12
human [11]	Human gene regulatory network	22	24.6	2214
kron [12]	Graph500, Synthetic Graph	262	21	156
msdoor [11]	Mesh of 3D object	415	20.2	97.3

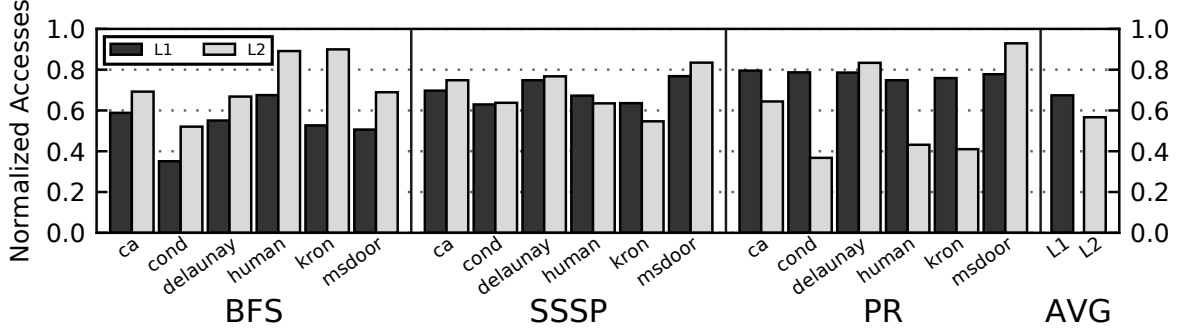


Figure 11: Normalized accesses to L1 and L2 caches of the IRU enabled GPU system against the Baseline GPU system. Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset.

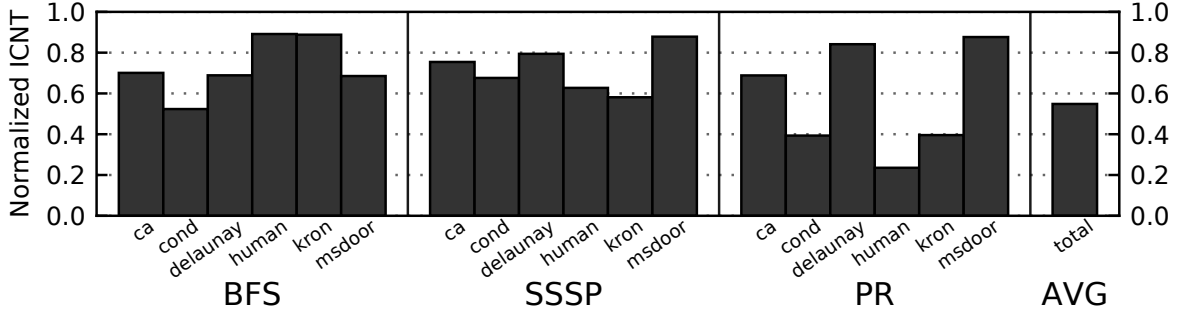


Figure 12: Normalized interconnection traffic between SM and MP of the IRU enabled GPU system against the Baseline GPU system. Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset.

6.1 Memory Pressure Reduction

IRU’s main functionality is to reorder irregular accesses improving their memory coalescing and, as a consequence, reducing the overall contention in the memory hierarchy. Figure 11 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, and overall accesses are reduced to 67% and 56% of the original L1 and L2 accesses on average.

This important reduction comes from several factors. First, the IRU reordering of irregular accesses improves coalescing which reduces the accesses to L1. Second, IRU reorders requests across SMs so it helps to collocate accesses of a particular memory block to a single SM, avoiding data replication across L1 data caches, resulting in improved hit ratios. Third, overall reduced accesses to L1 reduce capacity and conflict misses improving data thrashing and consequently reducing L2 accesses. Finally, IRU filtering further reduces accesses by removing or merging duplicated elements already processed in the IRU, which will not contribute to additional accesses unlike in the baseline.

L2 accesses reduction is greater than in L1 in some bench-

marks for SSSP and PR graph algorithms. A significant amount of the indices reordered by the IRU on SSSP and PR are used for irregular accesses performed by atomic instructions. GPGPU-Sim models incoherent L1 caches; atomic operations bypass the L1 and are handled at the L2 on the corresponding memory partition. IRU coalescing and filtering improvement for these operations does not reduce L1 accesses but L2 accesses, explaining the bigger 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 in it access different parts of the 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 12 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 achieving a reduction to 54% of the original interconnection traffic. 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 ac-

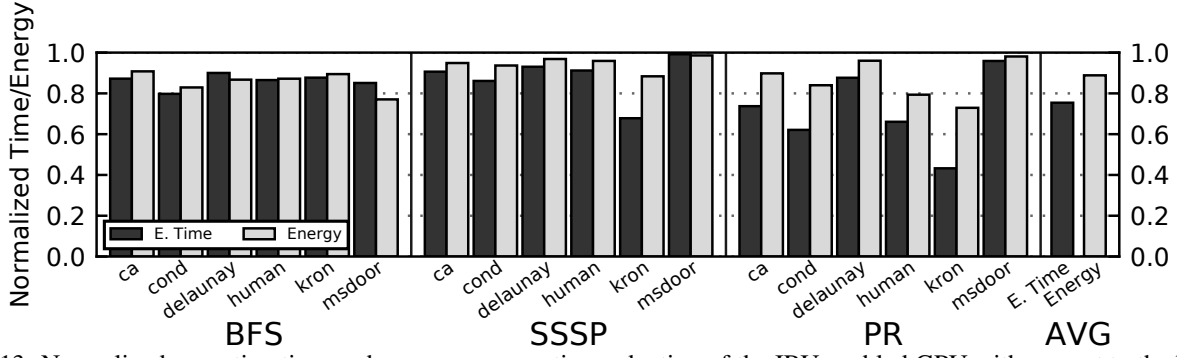


Figure 13: Normalized execution time and energy consumption reduction of the IRU enabled GPU with respect to the baseline GPU system. Significant speedups and energy savings achieved across BFS, SSSP and PR graph algorithms and every dataset.

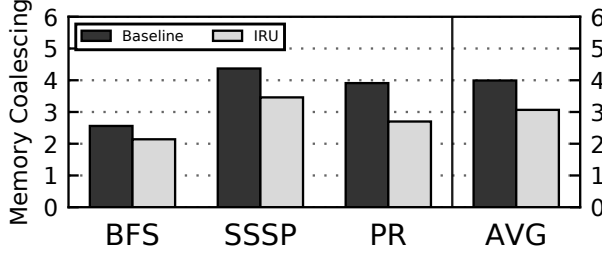


Figure 14: Improvement in memory coalescing achieved with the IRU over the Baseline GPU system. Vertical axis shows the number of memory requests sent to the L1 cache on average per each memory instruction, i.e. how many memory requests are required to serve the 32 threads in a warp.

cesses which reduces interconnection contention. Finally, the extended ISA instructions used on a `load_iru` operation allow reduced traffic by issuing a single request to the IRU that receives two replies (up to three replies), whereas the baseline GPU would have issued two requests and two replies in order to gather data in different frontiers.

Figure 14 shows the improvement in memory coalescing delivered by the IRU. A higher coalescing number indicates that more accesses are needed 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 memory coalescing for every graph algorithm from 4 to 3 accesses per memory requests on average, requiring one fewer access per request. This improvement is remarkably good given that the filtering schemes that some of the algorithms employ, combined with the filtering applied by the IRU, reduces the potential of coalescing in some degree, since filtering removes some duplicated elements. In spite of that, overall 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 very significant improvements in memory coalescing and reduce contention in every memory hierarchy component.

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. On average, 48.5% of the

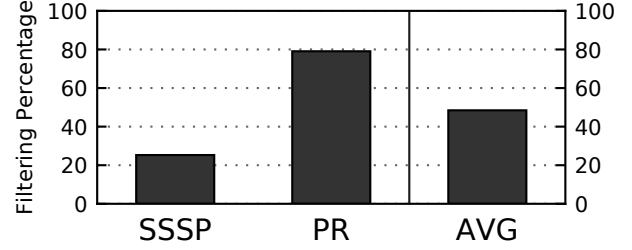


Figure 15: Filtered percentage of elements processed by the IRU in our IRU enabled GPU system. The IRU achieves significant filtering effectiveness across all graph algorithms.

elements are filtered by the IRU. This rather high percentage does not directly indicate that a similar amount of accesses to memory are discarded, yet it contributes on top of the IRU reordering improvement. This situation is explained due to its interaction with software graph filtering schemes. Some of the kernels instrumented already employ filtering schemes in their code, typical in graph-based applications, thus the filtering not always contribute significantly in memory accesses reduction. Yet, it efficiently filters elements avoiding evaluating the more costly software filtering schemes of graph algorithms, which improves overall performance.

6.3 Performance Evaluation

IRU provides performance improvement across all algorithms and benchmarks, as seen in Figure 13. 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 experiences higher speedups due to significantly larger reduction of L2 accesses due to the IRU filtering, which merges data and avoids costly atomic L2 accesses. SSSP achieves the lowest speedup due to lower filtering effectiveness.

Overall, performance improvements come from two sources. First, the improved memory coalescing due to the IRU reordering of indices used for irregular accesses, which reduces contention on the memory hierarchy. Second, the IRU filtering and merging that enables further reduction of accesses and redundant use of the Execution Units of the GPU.

6.4 Energy Evaluation

Figure 13 also shows the energy savings achieved with 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 per-

formance improvements since the IRU greatly reduces L1 and L2 accesses but achieves a more modest reduction of main memory accesses, main memory representing a very significant portion of the total energy consumed. The IRU energy overhead represents a small 0.5% of the final energy.

Overall, energy savings are obtained from several sources. First, the reduced accesses to L1 and L2 and contention to 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 filtering than the costly filtering employed by the graph applications. Finally, the IRU reordering leads to a reduction in main memory accesses which contributes to the achieved energy reduction.

6.5 Area Evaluation

Our evaluation of the IRU energy and area estimations indicate that the IRU requires a total of 23.9 mm² when adding up all the 4 partitions of our GPU system with a GTX980, each partition being 5.98 mm². 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 minimal area requirements.

7. RELATED WORK

Irregular programs on GPGPU architectures face many challenges that often result 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 [6, 29, 33, 41]. Nonetheless, if these issues are overcome, irregular applications can greatly benefit of the high parallelism that GPU architectures offer. Consequently, 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 [20, 31] to improve utilization of execution units, by cooperatively processing data over thread warps and thread blocks, yet adding complexity to the algorithm. Others provide a more transparent architectural approach at remapping threads over warps to improve branch divergence [13]. Some memory divergence approaches propose modifying software data structures [15, 32, 40]. 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]. The majority of these works propose software methods requiring extensive programming effort to change algorithm behavior and data structures, profound hardware knowledge or exploiting application characteristics. 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 involvement

to achieve the desired result. Extensive research has been done on flexible cache solutions [16, 23, 25] that realize the different accesses granularity of irregular memory accesses and propose dynamic cache organizations that adapt for fine-grained and coarse-grained accesses, improving L1 cache utilization and reducing contention. The same contention reduction objective is pursued by other works by resorting to cache bypassing mechanisms [7]. Other works propose hybrid software and hardware approaches that enable data dependent aware dynamic scheduling [43]. Finally, works such as D2MA [18] and Stash [21] 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 which is poor memory coalescing that leads to high memory contention. Our approach reduces divergence still present for global accesses performed with scratchpad managing mechanisms.

Intermediate approaches have explored extending the GPU architecture with custom purpose hardware units. SCU [38] 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 not tailor made to improve a particular problem (i.e stream compaction operations) but general irregular accesses patterns. Furthermore, the SCU requires significant changes in the application, since entire CUDA 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 Section 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 of GPU irregular program execution and exploit deep knowledge of graphs data structures to propose near data processing approaches. Proposals include standalone approaches such as TuNao [45], Dram based Graphicionado [17], PIM-based GraphH [9] or GraphQ [46] and SSD based approaches such as GraphSSD [30]. 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 optimization delivers 1.32x improved memory coalescing, significantly reducing by 46% the traffic in the memory hierarchy. 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.

REFERENCES

- [1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.
- [2] S. Beamer, "Understanding and improving graph algorithm performance," Ph.D. dissertation, UC Berkeley, 2016.
- [3] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.
- [4] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.
- [5] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the conference on high performance graphics 2009*, 2009, pp. 159–166.
- [6] M. Burtcher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.
- [7] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 343–355.
- [8] J. Chong, E. Gonina, and K. Keutzer, "Efficient automatic speech recognition on the gpu," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 601–618.
- [9] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.
- [10] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 349–359.
- [11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.
- [12] DIMACS, "10th dimacs implementation challenge - graph partitioning and graph clustering," 2010. [Online]. Available: <https://www.cc.gatech.edu/dimacs10/>
- [13] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 407–420.
- [14] A. Geil, Y. Wang, and J. D. Owens, "Wtf, gpu! computing twitter's who-to-follow on the gpu," in *Proceedings of the second ACM conference on Online social networks*. ACM, 2014, pp. 63–68.
- [15] A. Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sallinen, and M. Ripeanu, "Efficient large-scale graph processing on hybrid cpu and gpu systems," *arXiv preprint arXiv:1312.3018*, 2013.
- [16] H. Guo, L. Huang, Y. Lü, S. Ma, and Z. Wang, "Dycache: Dynamic multi-grain cache management for irregular memory accesses on gpu," *IEEE Access*, vol. 6, pp. 38 881–38 891, 2018.
- [17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphiconado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.
- [18] D. A. Jamshidi, M. Samadi, and S. Mahlke, "D2ma: Accelerating coarse-grained data transfer for gpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 431–442.
- [19] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPs)*. IEEE, 2018, pp. 287–296.
- [20] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 39–50.
- [21] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 707–719, 2015.
- [22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [23] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 376–388.
- [24] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwatch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41. ACM, 2013, pp. 487–498.
- [25] B. Li, J. Sun, M. Annavaram, and N. S. Kim, "Elastic-cache: Gpu cache architecture for efficient fine-and coarse-grained cache-line management," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 82–91.
- [26] J. Li, S. Ranka, and S. Sahni, "Strassen's matrix multiplication on gpu," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 157–164.
- [27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.
- [28] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.
- [29] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.
- [30] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "Graphssd: graph semantics aware ssd," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 116–128.
- [31] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Transactions on Parallel Computing (TOPC)*, vol. 1, no. 2, pp. 1–30, 2015.
- [32] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 622–636, 2018.
- [33] M. A. O'Neil and M. Burtcher, "Microarchitectural performance characterization of irregular gpu kernels," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 130–139.
- [34] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix-vector multiplication using reordering techniques on gpus," *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 65–77, 2012.
- [35] C. Rivera, J. Chen, N. Xiong, S. L. Song, and D. Tao, "Ism2: Optimizing irregular-shaped matrix-matrix multiplication on gpus," *arXiv preprint arXiv:2002.03258*, 2020.

- [36] C. Root and T. Mostak, "Mapd: a gpu-powered big data analytics and visualization platform," in *ACM SIGGRAPH 2016 Talks*, 2016, pp. 1–2.
- [37] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.
- [38] A. Segura, J.-M. Arnau, and A. González, "Scu: a gpu stream compaction unit for graph processing," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 424–435.
- [39] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, vol. 1, no. 1, pp. 1–17, 2008.
- [40] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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*, 2016, pp. 1–12.
- [41] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 140–149.
- [42] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.
- [43] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, Y. Xiaochun, Z. Zhimin, F. Dongrui, and X. Yuan, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.
- [44] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, and M. J. Franklin, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
- [45] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "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)*. IEEE, 2017, pp. 731–734.
- [46] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.