© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/TC.2021.3104749

# Energy-Efficient Stream Compaction Through Filtering and Coalescing Accesses in GPGPU Memory Partitions

Albert Segura, Jose-Maria Arnau, and Antonio González, *Fellow, IEEE*

**Abstract**—Graph-based applications are essential in emerging domains such as data analytics or machine learning. Data gathering in a knowledge-based society requires great data processing efficiency. High-throughput GPGPU architectures are key to enable efficient graph processing. Nonetheless, irregular and sparse memory access patterns present in graph-based applications induce high memory divergence and contention, which result in poor GPGPU efficiency for graph processing. Recent work has pointed out the importance of stream compaction operations, and has proposed a Stream Compaction Unit (SCU) to offload them to a specialized hardware. On the other hand, memory contention caused by high divergence has been tackled with the Irregular accesses Reorder Unit (IRU), delivering improved memory coalescing.
In this paper, we propose a new unit, the IRU-enhanced SCU (ISCU), that leverages the strengths of both approaches. The ISCU employs the efficient mechanisms of the IRU to improve SCU stream compaction efficiency and throughput limitations, achieving a synergistic effect for graph processing. We evaluate the ISCU for a wide variety of state-of-the-art graph-based algorithms and applications. Results show that the ISCU achieves a performance speedup of 2.2x and 90% energy savings derived from a high reduction of 78% memory accesses, while incurring in 8.5% area overhead.

**Index Terms**—Graph processing, Stream Compaction, GPGPU architectures, Memory divergence, Stream Compaction Unit (SCU), Irregular accesses Reorder Unit (IRU).

## 1 Introduction

Graph-based applications are ubiquitous in important domains such as data analytics [1] or machine learning [2] among many other examples. Road navigation and self-driving cars [3], recommendation systems [4] and speech recognition [5] are paradigmatic examples of graph processing workloads. Current trends towards increased data gathering [6] and knowledge-based applications result in an increased importance of graph-based applications and, at the same time, a demand for higher data processing capabilities, which motivates high-throughput graph processing on GPGPU architectures.

GPGPUs achieve high performance for regular programs that exhibit low branch and memory divergence. Unfortunately, graph algorithms exhibit sparse memory accesses, as they traverse unstructured and irregular data with unpredictable patterns [7]. Hence, GPGPU implementations of graph algorithms show significant memory divergence [8], which leads to high contention in the memory hierarchy and poor utilization of the functional units. Not surprisingly, some recent work focused on improving graph processing on GPGPUs through software-level optimizations [9], [10], [11]. Furthermore, graph frameworks such as Gunrock [12], nvGRAPH [13], HPGA [14] or MapGraph [15] have been introduced in recent years. Despite all these efforts, we found that state-of-the-art CUDA implementations still suffer from high contention in the memory hierarchy and low utilization of the functional units, as low as 13.5% on average in our graph datasets.

• *The authors are with the Department of Computer Architecture, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain. E-mail: asegura, jarnau, antonio@ac.upc.edu.*

*Manuscript received July 15, 2020*

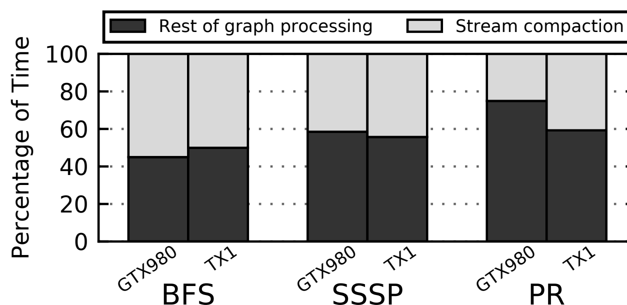


Fig. 1: Execution time breakdown for several applications and three graph primitives (BFS, SSSP and PR). Measured on an NVIDIA GTX 980 and Tegra X1.

One of the most effective optimization for GPGPU graph processing is stream compaction [11]. This technique is based on the observation that, on each iteration of a graph algorithm, only a small and sparsely distributed subset of the nodes/edges are typically active. Stream compaction gathers the data of active nodes/edges on a compacted array in contiguous memory, so subsequent processing on the compacted array exhibits much more regular memory access patterns and, hence, it runs efficiently on the Streaming Multiprocessors (SMs) of the GPU. However, the GPU is ineffective at performing stream compaction operations, which represent a large fraction of execution time across several graph algorithms, as shown in Figure 1.

Recognizing the importance of stream compaction in graph processing, recent work [8] proposes to offload this operation

TABLE 1: Comparison between SCU, IRU and ISCU hardware extensions for Graph Processing on GPGPU architectures.

| | GPU | GPU+SCU | GPU+IRU | GPU+ISCU |
|---|---|---|---|---|
| Offloaded task | Nothing | Stream Compaction | Irregular Loads | Stream Compaction |
| Speedup over GPU | - | 1.37x | 1.33x | 2.2x |
| Energy savings over GPU | - | 84.7% | 13% | 90% |
| Area overheads | - | 3.3% | 5.6% | 8.5% |
| NoC contention | High | High | Low | Low |
| Node/Edge filtering | In software (expensive) | In SCU (data in L2) | In L2 | In L2 |

to a specialized unit tightly integrated in the GPU, called the Stream Compaction Unit (SCU). The SCU is a small unit with an architecture tailored to the irregular memory accesses that arise in stream compaction. The remaining steps of the graph-based algorithm are executed on the SMs taking benefit of the large amount of parallelism in the GPU, but they operate on the SCU-prepared data and achieve much higher efficiency. The SCU writes the compacted nodes/edges in an order that improves memory coalescing and, in addition, it performs filtering of repeated and already visited nodes during the compaction process, significantly reducing GPGPU workload.

Filtering duplicated and already visited elements is key for high-performance graph processing. Software based solutions require expensive atomic operations to accurately detect repeated elements, or they loosely track duplicated nodes which severely impacts the effectiveness of the filtering. The SCU employs a hash table, stored in the L2 cache, to track already processed nodes/edges. According to our measurements, a large amount of data is moved between the SCU and the L2 partitions just for the filtering. More specifically, we have measured that 57% of the traffic in the NoC is due to the filtering operations of the SCU. We claim this is an important limitation and we improve the SCU design in this paper to avoid this bottleneck.

Another GPU architectural extension for graph processing, the Irregular accesses Reordering Unit (IRU), shows a more efficient design for filtering. Instead of offloading the entire stream compaction, the IRU focuses on improving coalescing of irregular memory accesses by reordering the node/edge frontier on-the-fly, so threads within the same warp receive nodes/edges stored in the same cache line. During this reordering, the IRU filters duplicated elements but, unlike the SCU, it is located inside the GPU memory partitions and it performs the filtering directly in the L2, reducing traffic in the NoC by a large extent.

Table 1 summarizes both approaches, SCU and IRU. The SCU achieves significant speedups and large energy savings, but it produces high contention in the NoC since a lot of data is moved between the L2 and the SCU for the filtering operation. On the other hand, the IRU achieves more modest energy savings, since it focuses on irregular load operations while most of the stream compaction still runs on the GPU, but its filtering operation is highly efficient as it is done inside the memory partitions, largely reducing contention in the NoC. The IRU requires 46% lower NoC traffic according to our measurements.

In this paper, we show that the IRU and the SCU have interesting synergies and we propose a novel GPU design that effectively combines both techniques. In our scheme, we leverage the SCU to offload the stream compaction operation. However, we modify the behavior of the filtering operation. Instead of fetching data from L2 and performing the filtering in the SCU, our SCU issues requests to the IRU to filter repeated elements in the memory partitions. In this manner, the IRU ameliorates the main bottleneck of the SCU, achieving the benefits of both solutions: large energy savings due to offloading stream compaction to a specialized unit and low contention in the NoC since filtering of duplicated elements is done inside the memory partitions. We call this system the IRU-enhanced SCU (ISCU).

This paper focuses on improving the performance of graph processing on GPGPU architectures. Its main contributions are the following:

- We characterize the bottlenecks of the SCU. We observe that the main limiting factor is the large amount of data movement between the L2 cache and the SCU required for the filtering operation, which represents 57% of NoC traffic.
- We identify the synergies between the SCU and the IRU, and show that they perfectly complement each other. Based on this observation, we propose the ISCU, a novel GPU extension that combines both the efficient SCU and the filtering mechanism of the IRU to improve overall graph processing efficiency.
- We evaluate our proposal on top of a modern GPU architecture. Our experimental results show that the ISCU improves performance by 2.2x and delivers 90% energy savings for a diverse set of graph-based applications over a GTX 980 GPU. Compared to the GPU+SCU, our ISCU improves performance by 63%, while achieving 66% energy savings.

The remainder of this paper is organized as follows. Section 2 reviews graph processing on GPGPU architectures and introduces the SCU and IRU hardware extensions. Section 3 presents the architecture of the ISCU, whereas Section 4 describes its API and programmability. Section 5 presents the evaluation methodology and Section 6 provides the experimental results. Section 7 reviews the related work and, finally, Section 8 sums up the main conclusions.

## 2 GPGPU Graph Processing

Graph processing is used in many domains and applications that employ graphs to represent data, such as road navigation systems or data analytics. Graphs consist of two main elements: nodes and edges. Graph nodes are used for data entries while the edges indicate relationships between the nodes of the dataset. The graph dataset is typically stored in a Compressed Sparse Row (CSR) [9] format which reduces memory footprint. The characteristics of graph traversal algorithms are highly dependent on the topology of the graph, which tends to be highly unstructured and irregular, resulting in unpredictable memory access patterns. These characteristics negatively impact graph exploration efficiency on GPGPU architectures. Nonetheless, graph processing is highly parallelizable, as many elements can be processed simultaneously without dependencies, which GPGPU architectures can exploit with their huge parallelism.

Graph traversal algorithms on GPGPU architectures consist of an iterative process. Each iteration starts with the set of active nodes, i.e. the node frontier. Each element in the node frontier is processed in parallel. Afterwards, edges departing from active nodes are traversed to generate the node frontier for the next iteration. This process is repeated until the algorithms converges, i.e. it reaches the stop condition. GPGPU architectures assign a given element of the frontier to a thread. Since graph exploration happens in parallel, synchronization mechanism are used to avoid expanding duplicated nodes, which would severely increase workload.

In this work we focus on popular graph algorithms, in particular Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP) and PageRank (PR). BFS computes the shortest hops from a source node to all the other nodes in the graph. SSSP computes the shortest distances from a source node to any node (i.e. every edge having a different weight). PR computes a ranking of the most well interconnected nodes in a graph which is used in recommendation systems. The state-of-the-art CUDA implementation of BFS [10] uses best-effort synchronization approaches, that avoid the use of atomic operations but provide a degree of filtering of duplicated elements. SSSP [16] and PR [4] use atomic operations to avoid duplication, yet incur in significant overheads.

Taking BFS as an example, its exploration consists of two main phases or kernels: Expansion and Contraction. In the Expansion kernel, a thread processes a node in the node frontier expanding its edges into the edge frontier, a process done cooperatively with other threads to improve thread balancing, as some nodes might have higher connectivity than others. In the Contraction kernel, a thread is used per edge in the edge frontier, if the destination node of each edge has not been visited it sees its hops from the source node updated, is marked as visited and is inserted in the next node frontier, otherwise it is not processed. As BFS uses best-effort filtering of visited nodes it might expand nodes already visited, increasing workload, but producing a correct result. Further details about SSSP and PR graph processing phases can be found in the SCU paper [8].

### 2.1 Graph Processing and SCU Hardware Extension

Stream compaction operations improve memory coalescing and locality of sparse accesses performed by graph applications. However, this process is inefficiently performed in the GPU due to the high amount of irregular accesses and synchronization overheads which leads to inefficient use of the GPU resources. Although stream compaction results in a net performance improvement, it represents about 50% of the total execution time of graph applications [8]. The Stream Compaction Unit (SCU) is a GPU hardware extension tailored to efficiently perform compaction operations which are offloaded from the GPU, this provides high performance and energy efficient hardware by avoiding both synchronization overheads and utilizing the GPU to perform inefficient data movements with main memory.

#### 2.1.1 SCU Architecture Overview

The SCU is a new hardware unit added to the GPU, which is attached to the interconnection of the GPU as shown in Figure 2. The internal pipeline of the SCU consists of a set of components that are able to efficiently issue, gather and compact sparse irregular data, while providing flexibility to allow for a comprehensive set of data operations, detailed in Section 4.

The host-issued SCU operations configure the unit and initiate its processing. First, the *Address Generator* component obtains

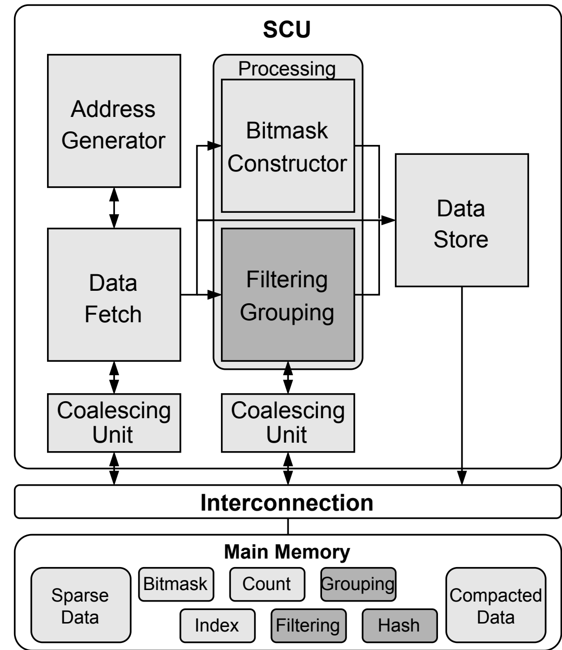


Fig. 2: SCU internal pipeline and memory data structures.

parameters and issues requests only to the data to compact from memory, which then is fetched by the *Data Fetch* component from main memory through the GPU L2. Afterwards, the fetched data is processed in the *Processing* component according to the operation launched, this unit provides the functionality to perform complex data compaction operations and optimizations. Finally, the resulting data is then forwarded to the *Data Store* which writes directly to memory in a compacted manner.

As indicated, the SCU can perform data optimizations by filtering and grouping pre-processing, this can be done more efficiently than in the GPU when performing the gathering of data. To deliver these optimizations, the SCU uses an in-memory hash that is cached in GPU L2, this is done to to reduce hardware overheads, as well as allowing for re-configuration to tailor the data optimizations. For the filtering operation, the hash table provides a low-cost mechanism to loosely remove duplicates. Each new edge/node probes the hash table and is discarded if a previous occurrence of the same node/edge is found. In case of hash collisions the corresponding hash table entry is overwritten, which means that false negatives are possible. Nonetheless, SCU provides a highly effective filtering reaching more than 70% of the workload [8] without incurring in significant overheads. On top of this, it removes additional synchronization overheads required for regular filtering steps performed in successive steps the GPU, that are required for software-based solutions for removing duplicated nodes/edges. For the grouping operation, the hash table is reconfigured to create groups of edges whose destination node lies in the same cache line, in order to store them together in the compacted array.

#### 2.1.2 SCU Programming Model

The SCU is capable of performing a set of data compaction operations issued from the host CPU with an API detailed in Section 4. This set of operation enables complex data compaction patterns which are performed by GPU compaction algorithms.

Each SCU operation uses a set of parameter vectors stored in main memory which are provided by the CPU and are fetched when an operation is started. These parameters are shown in Figure 2, they are used to indicate the sparse data to fetch with the *bitmask*, indirections and duplications to perform with the *index* and *count* and to enable the data optimizations with the *filtering* and *grouping* vectors.

The BFS algorithm [10] is instrumented in Figure 3 in order to use the SCU with operations shown in Section 4. Both BFS kernel data compaction efforts can be offloaded using the SCU stream compaction operations, meanwhile the GPU performs the regular part of the graph exploration. In the first kernel *BFS_Expand*, the SCU efficiently compacts in memory all the edges from the active nodes processed, while in the second kernel *BFS_Contract* it efficiently filters out the discarded edges.

```
1  // nF: node_frontier, eF: edge_frontier
2  void BFS_Expand (nF) {
3      indexes,count = BFS_preparationGPU (nF);
4      eF = accessExpansionCompactionSCU
5          (edges, indexes, count);
6      return eF;
7  }
8
9  void BFS_Contract (eF) {
10     bitmask = BFS_contractionGPU (eF);
11     nF = dataCompactionSCU (eF, bitmask);
12     return nF;
13 }
```

Fig. 3: Pseudo-code of BFS using the SCU Hardware Extension.

### 2.2 Graph Processing and IRU Hardware Extension

Irregular accesses performed by graph applications have high memory divergence which increases overall use of memory hierarchy resources and reduces data locality. GPGPU coalescing hardware targets an application access patterns which are often hard to optimize by the programmers, requiring major code overhauls, yet memory divergence remains high requiring for memory requests per group of threads. The Irregular Accesses Reorder Unit (IRU) hardware extension transparently improves divergence in irregular accesses performed by the GPU. The IRU achieves this by reordering indices in the node/edge frontiers, an optimization made possible by relaxing the GPU programming model restriction as to allow threads to retrieve reordered indices, which is valid in graph applications since threads can process any given element.

#### *2.2.1 IRU Architecture Overview*

The IRU is a GPU hardware extension placed in the memory partitions of the GPU alongside other components it it, as shown in Figure 4. Each IRU partition consists of a set of components responsible to fetch and reorder the indexes used in irregular accesses by the GPU, thus reducing memory divergence. The IRU reordering is enabled by the use of a partitioned hash table, which is employed to gather indices that will collocate to the same memory block. The direct connection of the IRU partitions allows a global scope when reordering indexes across all SMs, for which a regular GPU system is not capable and is key on further reducing divergence.

Initially, the IRU is configured from the host by receiving information of which indices are targeted for divergence improvement.

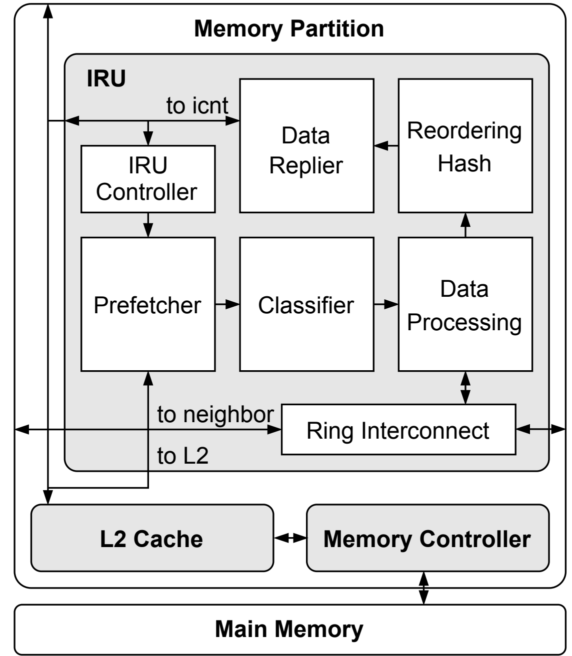


Fig. 4: IRU internal pipeline in a Memory Partition.

Upon the kernel being launched, the different IRU partitions start prefetching the indices with a *Prefetcher*. These indices are later classified and issued to the local *Reordering Hash* or sent to the *Ring Interconnection* if they target another hash partition. Indices are collocated into the hash table until a request from the IRU ISA instructions arrives at the partition, which is serviced with the reordered indices that will result in improved memory coalescing in subsequent memory accesses performed by the GPU. Additionally, the hash is also able to filter duplicated indices, which disable the execution of the requesting threads reducing workload.

The *Reordering Hash* is direct mapped and multi-banked, each entry holding 32 indexes which are filled consecutively at each insertion. The indexes in an entry target the same memory block, reducing memory divergence. The hash allows collocating indexes that do not match tags thus accessing different memory blocks, a feature which allows to reduce conflict handling complexity. Nonetheless, the amount of conflicts is mitigated with a dispersion hash function and sufficiently sized hash. Note that some of these conflicting elements might collocate among themselves, thus not severely impairing memory coalescing.

#### *2.2.2 IRU Programming Model*

The IRU introduces new ISA instructions and API which requires very simple changes to the code, further details are provided in Section 4. The operations provided replace regular load operations with a request to the IRU for that same data. Additional host operations are provided to configure the IRU with the indexes targeted for reordering. Figure 3 shows a BFS GPGPU kernel instrumented to use the IRU showcasing the simple instrumentation required.

## 3 IRU-ENHANCED SCU (ISCU)

In this section we present the IRU-enhanced SCU (ISCU), a GPGPU hardware extension targeting graph processing applications. The ISCU improves the SCU by utilizing the IRU hardware

```
__global__ void BFS_contractionGPU (...) {
  int pos = blockDim.x * blockIdx.x +
            threadIdx.x;
  if (pos < number_elements) {
    int edge;

#ifdef NOT_INSTRUMENTED
    edge = edge_frontier[pos];
#elif USE_IRU
    load_iru(edge);
#endif

    // additional computations ...
    label[edge] = distance;
  }
}
```

Fig. 5: Instrumentation of a BFS algorithm Kernel using the IRU.

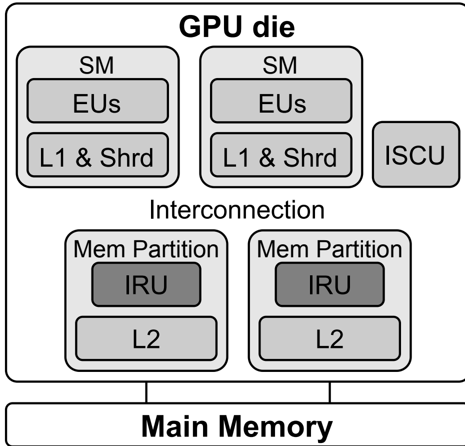


Fig. 6: GPGPU architecture including the ISCU extension.

extension to perform pre-processing operations, in particular the filtering of duplicated nodes/edges. The ISCU combines the powerful SCU optimizations obtained by offloading stream compaction operations with the efficient hashing mechanism used in the IRU. Figure 6 showcases a GPGPU architecture featuring the ISCU and IRU partitions located in the GPGPU Memory Partitions (MP). The ISCU extension is motivated by SCU's bottleneck experienced when performing pre-processing filtering and grouping optimizations for graph processing applications.

### 3.1 SCU and IRU Synergies

SCU's main bottleneck arises from the limited interconnection throughput to the L2 and due to the memory accesses required to perform filtering/grouping operations through the in-memory hash table. Figure 7 shows the utilization of the filtering/grouping unit, measured as the percentage of cycles this data pre-processing unit is being utilized over the total execution, and the percentage of Network-on-Chip (NoC) traffic generated by it. Utilization of this component is high during the execution of the compaction operations, reaching 92% of the execution for BFS and an average of 51% for the different graph algorithms. Furthermore, it is responsible for a significant amount of traffic and accesses to the interconnection, as much as 80% for BFS and an average of 58% for the different graph algorithms. Consequently, this component's high utilization of the pipeline and NoC limits performance and provides an opportunity for optimization.

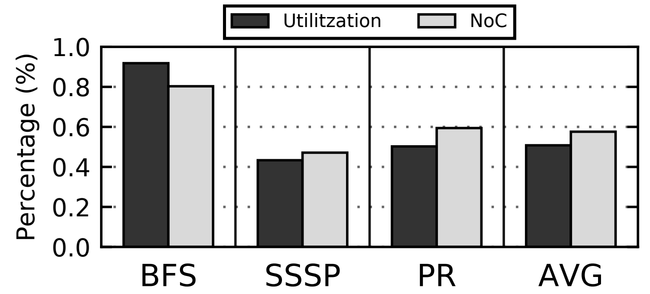


Fig. 7: Utilization of the SCU pre-processing component (i.e. filtering/grouping unit) and the percentage of NoC traffic devoted to filtering/grouping operations. The utilization is the percentage of cycles that the filtering/grouping unit is active over total execution. The SCU invests a large number of cycles and NoC transactions in the data pre-processing operations.

The memory accesses that saturate NoC throughput come from several sources. First, from fetching the sparse data and then writing the elements in the compacted array. Second, from the parameters used in the operations. Finally, when doing pre-processing, several accesses are required to retrieve and operate with the in-memory hash table. Nonetheless, the high filtering efficiency achieved, reaching up to 76% of the workload, reduces significantly the accesses required for the data compaction operations themselves, consequently accesses to the in-memory hash table represent a larger split and become a significant bottleneck.

Insertion of elements to the in-memory hash table requires several accesses. For filtering, first it requires fetching the tag entry, and performing the corresponding comparison. In case of a miss, tag and data entries have to be updated. Consequently, processing an element (edge or node) incurs in multiple accesses to the L2. Although the SCU in-memory hash table design is multi-banked, the throughput to L2 is limited to a single access from the *Filtering/Grouping* component per cycle, severely affecting the performance of hash insertions.

We propose to use the IRU efficient distributed hash table as a replacement of the in-memory hash used for the SCU pre-processing, additionally increasing the throughput of requests to the IRU. The resulting system that we term ISCU contains both SCU and IRU hardware extensions with our modifications to fit the requirements of the end system.

Finally, ISCU synergistic use of both SCU and IRU systems not only addresses the SCU overheads previously mentioned but also enable further improvement of targeted graph processing GPGPU algorithms. The ISCU allows to perform more efficient data compaction utilizing the whole ISCU hardware, while reducing memory divergence with the IRU.

### 3.2 Hardware Enhancements

The main changes in the SCU are modifications to the *Data Fetch* and *Filtering/Grouping* component shown in Figure 2. Due to the data path changes reviewed in Section 3.3, *Data Fetch* is only required to issue the fetch operation, yet the IRU is the hardware receiving that data, avoiding unnecessary data movements. Similarly, the *Filtering/Grouping* component logic is largely removed since it was responsible to manage request to the in-memory hash table. Additionally, the coalescing unit attached

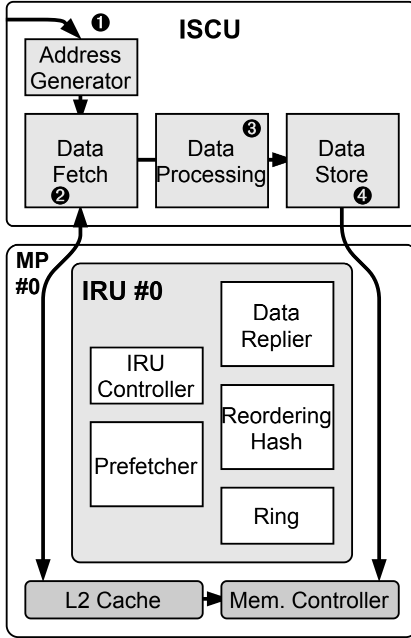


Fig. 8: Behavior and data-flow of a regular ISCU operation on the ISCU hardware extension.

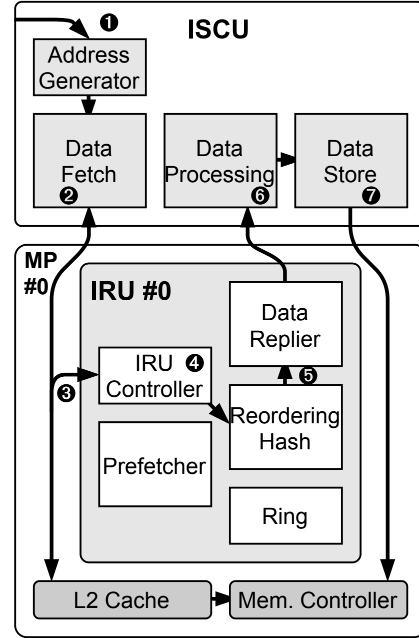


Fig. 9: Behavior and data-flow of a pre-processing ISCU operation, i.e. filtering/grouping.

to this component is no longer required as it was used to merge request to tag and data entries.

Similarly, adapting the IRU requires minor hardware changes. Additional control logic is required to support configuring the IRU to perform SCU pre-processing. This control logic modifies the data-path, so compaction operation are performed at different locations in the system: the SCU will initiate the requests to the data, but the replies from the memory controller will be directly passed to the IRU, that is located inside the L2 partition. In this manner, data can be filtered and reordered in the IRU without being transferred to the SCU through the NoC, saving NoC bandwidth by a large extent. In our system, the IRU does not use the prefetcher to gather data, since the SCU is in charge of orchestrating the stream compaction operation and it takes care of generating the read requests to the memory controller. Additionally, the *Data Replier* does no longer require to gather requests, and can send replies back directly when data from the *Reordering Hash* is ready. Although the *Prefetcher* is not utilized for the ISCU, this structure is maintained to provide the IRU improvements in other kernels.

The hash table mechanism of the original SCU is not bound by on-chip memory size as it is stored in main memory and cached in the L2. In comparison, the IRU includes limited on-chip storage for the hash table. This reduces memory bandwidth usage at the cost of less accurate filtering of duplicated nodes, since in case of conflicts in the hash table the old data is evicted. We have observed that with a modest size of 80KB per memory partition the filtering mechanism is highly effective, as it is able to avoid the vast majority of duplicated elements.

## 3.3 Detailed Data Processing

The ISCU has two main internal processing data-flows which are represented in Figure 8 for regular operations, and in Figure 9 for data pre-processing, both corresponding to the different operations listed in Section 4.

### 3.3.1 Regular ISCU operations

The internal processing and data-flow of regular ISCU operations is shown in Figure 8. Initially, an operation is issued from the host which configures the ISCU with the required parameters and starts the execution ❶. This initializes the Address Generator to fetch parameters and start fetching from L2 the sparse data to compact ❷. Afterwards, according to the corresponding operation, some processing is applied to the data, such as replication or indirection ❸. Finally, the sparsely gathered data is compacted and written directly to main memory ❹.

### 3.3.2 Pre-processing ISCU operations

The behavior and data-flow of pre-processing ISCU operations is shown in Figure 9. The initial configuration ❶ and fetching of the sparse data ❷ is done the same way as regular ISCU operations. Additionally, the ISCU pre-processing operation configures the *IRU Controller* to receive and process data from the ISCU.

The processing and data-flow changes with respect to the SCU start in the *Data Fetch* component. The data fetched by the ISCU is directly sent to the IRU ❸. In this manner, duplicated and already visited nodes/edges are not transferred to the ISCU since they are removed in the memory partition, saving NoC bandwidth by a large extent. The elements used for pre-processing are then forwarded to the corresponding IRU through the *Ring*, if the hashing function dictates it. Afterwards, these elements are inserted into the hash table performing the corresponding filtering or reordering operation ❹. When a hash entry is ready or no more data is to be inserted, the pre-processed data is forwarded to the *Data Replier* ❺, which sends a reply to the ISCU. Since the resulting pre-processed data has to be written to main memory, it might be destined to a different memory partition and so the ISCU handles the final writing. Finally, the ISCU creates the corresponding filtering/grouping vectors ❻ which are then written in memory directly by the *Data Store* ❼.

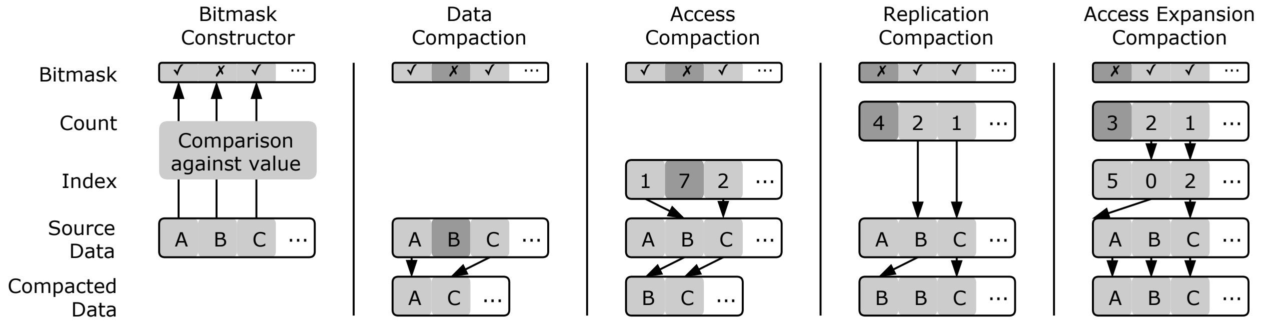


Fig. 10: SCU operations required to implement stream compaction capabilities, illustrated with the data that each operation uses and generates. Arrow direction indicates flow of data.

## 4 ISCU PROGRAMMABILITY

The modifications introduced to the ISCU hardware are architectural so the programming model remains unaffected. This section describes the complete ISCU programming model with the SCU and IRU operations and shows how graph applications can be instrumented to utilize the new hardware.

### 4.1 ISCU Compaction Operations

The compaction operations supported by the ISCU are listed in Figure 10. These operations have several parameters which are omitted in the figure for the sake of simplicity: the size of the data and the number of elements on which to operate. The ISCU implements the following operations:

- **Bitmask Constructor:** Generates a bitmask vector used by other operations. It requires a reference value and a comparison operation. It creates a bitmask vector for which each bit is set to one if the element in the input data evaluates to true, using the comparison operator and the reference value, and to zero otherwise.
- **Data Compaction:** Accesses sparse data sequentially and filters out the unwanted elements using the bitmask vector. The output at the destination contains only valid elements preserving the original order.
- **Access Compaction:** Accesses a sparse index vector sequentially and filters out the unwanted elements with a bitmask vector. The output at the destination contains only valid elements preserving the original order.
- **Replication Compaction:** Extension of the Data Compaction operation, which operates with the count vector. This vector is used to indicate how many times each element in the sparse data will be replicated in the output destination. The output destination contains only the valid elements, but each element is replicated by the amount of times indicated by its corresponding counter.
- **Access Expansion Compaction:** Uses both the indexes and count vectors. It is an extension of the Access Compaction operation, that copies a number of consecutive elements instead of only one element from the sparse data indicated by the corresponding indexes vector entry. The number of elements to gather is determined by the corresponding entry in the count vector.

### 4.2 Graph Processing Instrumentation

We instrument state-of-the-art implementations of BFS, SSSP and PR to utilize the optimizations offered by the ISCU.

#### 4.2.1 Breadth-First Search ISCU instrumentation

Filtering out duplicated elements is beneficial for both the expansion and contraction phases of the BFS algorithm. Grouping is also applicable, but interferes with the warp culling filtering efforts done in the GPU processing, which lowers its effectiveness and results in increased workload, largely reducing the performance benefits due to the improved memory coalescing. Shown on Figure 11, the required changes are the following:

```
1  // nF: node_frontier, eF: edge_frontier
2  void BFS_Expand (nF) {
3      indexes,count = BFS_preparationGPU (nF);
4      eF = accessExpansionCompactionSCU
5          (edges, indexes, count, do_filter);
6      return eF;
7  }
8
9  void BFS_Contract (eF) {
10     bitmask = BFS_contractionGPU (eF);
11     node_frontier = dataCompactionSCU
12         (ef, bitmask, do_filter);
13     return nF;
14 }
```

Fig. 11: Pseudo-code of the additional operations for a GPGPU BFS program to use the ISCU.

The Expansion phase performs the filtering directly when processing the data, generating the filtered edge frontier. It does not require the use of the filtering vector, which would be employed to apply the filtering to multiple compaction operations.

The Contraction phase also performs the filtering directly when processing the data and generates the final filtered node frontier. Note that this filtering is applied because the filtering done by BFS is not complete (as in SSSP).

Furthermore, the IRU is used on its own to provide irregular accesses improvement to the *BFS_contractionGPU* kernel, as seen in Figure 5, where a simple one LOC modification achieves improved irregular access coalescing employing the IRU.

#### 4.2.2 Single-Source Shortest Paths ISCU instrumentation

Filtering out of duplicated elements is beneficial for both the expansion and contraction phases of the SSSP algorithm. Additionally, unlike BFS, the grouping does not interfere with the GPU filtering, and the coalescing improvement results in a net gain in performance. Figure 12 shows the following required changes.

For the Expansion phase two additional Access Expansion Compaction are required. One operation is responsible for constructing

```
// nF: node_frontier, eF: edge_frontier,
// wF: weight_frontier
void SSSP_Expand (nF) {
    indexes,count = preparationGPU (nF);
    filtering = accessExpansionCompactionSCU
        (edges,indexes,count,do_filter);
    grouping = accessExpansionCompactionSCU
        (edges,indexes,count,do_grouping);

    eF = accessExpansionCompactionSCU
        (edges, indexes, count, filtering,
         grouping);
    wF = accessExpansionCompactionSCU
        (weights, indexes, count, filtering,
         grouping);
    wF += replicationCompactionSCU
        (weights, count, filtering,
         grouping);
    return eF, wF;
}

void SSSP_Contract (eF, wF, threshold) {
    bitmask_near,bitmask_far =
    SSSP_contractionGPU (eF, wF, threshold);
    grouping = dataCompactionSCU
        (eF, bitmaskNear);

    node_frontier = dataCompactionSCU
        (eF, bitmask_near, grouping);
    farPileEdges = dataCompactionSCU
        (eF, bitmask_far, grouping);
    farPileWeights = dataCompactionSCU
        (wF, bitmask_far, grouping);
    return nF;
}
```

Fig. 12: ISCU-enhanced SSSP with needed pseudo-code operations.

the filtering vector and the other for generating the grouping vector. The following operations use the previously generated vectors to filter and group the compacted data of the new edge frontier.

The first contraction phase operates on the "near" elements at each iteration of the algorithm. For this phase, only grouping is applicable, since the filtering done on the GPU is complete, and doing ISCU filtering would result in no benefit. The grouping information is only used by the subsequent operation that processes "near" elements, which result in the new grouped node frontier.

The second contraction phase operates on the "far" elements when there are no more "near" elements. For this phase both grouping and filtering are beneficial, since elements on the "far" pile are not filtered beforehand. Two additional Data Compaction operations are used to create the filtering and the grouping information for the "far" elements, which will be used by the subsequent operation that processes the "far" elements and generates the new filtered and grouped node frontier.

Furthermore, the IRU is used on its own to improve irregular memory accesses in the *SSSP_contractionGPU* kernel. The optimization is similar to the one performed for BFS in Figure 5, but additionally, the weight element and the original position are retrieved from the IRU.

#### 4.2.3 PageRank ISCU instrumentation

Removing duplicated or already visited nodes is not an option for PR since it requires to consider all the nodes' ranks on every iteration of the algorithm. The Update phase of the PR requires the use of atomic operations to correctly add the weights, a mechanism which is very costly, especially in large graphs. The filtering operation of the ISCU can be employed to compute the new ranks instead of using a large number of expensive atomic operations in the GPU. In other words, the filtering hardware in the IRU can be used to perform a reduction operation, adding the weights of duplicated nodes. Figure 13 shows the following required changes.

```
// eF: edge_frontier, wF: weight_frontier
void PR_Expand (nodes) {
    indexes, count = preparationGPU (nodes);
    filtering = accessExpansionCompactionSCU
        (edges, indexes, count);

    eF = accessExpansionCompactionSCU
        (edges, indexes, count, filtering);
    wF = replicationCompactionSCU
        (weights, count, filtering);
    return eF, wF;
}
```

Fig. 13: ISCU-enhanced PR with needed pseudo-code operations.

For the expansion phase we include an additional ISCU operation with the filtering mechanism that generates the filtering vector. Furthermore, the IRU is used standalone to provide irregular accesses improvement to other PR kernels. The optimization is similar to the one performed for BFS in Figure 5, but additionally, the weight element is retrieved from the IRU and the filtering is enabled to further reduce workload.

## 5 Evaluation Methodology

We model four different systems: the GPU (NVIDIA GTX 980 [17]), the GPU+SCU [8], the GPU+IRU and the GPU+ISCU presented in Section 3. To obtain GPU execution time we use GPGPU-Sim 3.2.2 [18], whereas GPU power, energy and area estimations are obtained with GPUWattch [19]. The GPU parameters used for the experiments are listed in Table 3, trying to accurately track the architecture of the NVIDIA GTX 980. To evaluate the SCU as presented in [8], we use a cycle-accurate simulator named SCU-sim, whereas we leverage a Verilog implementation to obtain SCU power and area. To obtain performance of the IRU, we extend the memory partitions in GPGPU-Sim as described in. IRU's area and power dissipation are estimated by using CACTI [20]. Finally, to evaluate our GPU+ISCU, we integrate both GPGPU-Sim with the IRU extension and SCU-sim. We model all the hardware changes described in Section 3. Figure 14 illustrates the simulation infrastructure employed in this paper.

The configuration parameters of the SCU are shown in Table 4. We set the SCU clock rate at 1.27 GHz in order to match the GPU frequency, and we configure the SCU to process 4 elements/cycle. We use a 5 KB FIFO to buffer the vector parameters of the SCU operations, while the *Data Fetch* component includes a 38 KB FIFO requests buffer and the *Filtering/Grouping* a 18 KB buffer. Finally, the coalescing units hold up to 32 in-flights requests with a merge window of 4 elements.

The SCU architecture is implemented in Verilog. In order to obtain area and energy consumption we synthesize the RTL code using the Synopsis Design Compiler [23] and the technology library of 32 nm from Synopsys with low power configured at 0.78V. Additionally, we use CACTI [20] to characterize cache and interconnection components.

Finally, we use the recent DRAMSim3 [24] main memory simulator, which provides improved and more accurate evaluation

TABLE 2: Diverse benchmark graph datasets collected from well-known repositories.

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

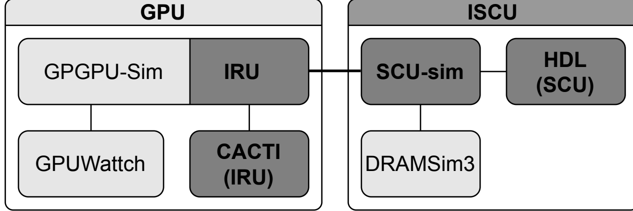


Fig. 14: ISCU complete simulation system comprising IRU-extended GPU simulation and SCU simulation. The darker color shows our contributions to the simulation system.

TABLE 3: GPGPU-Sim parameters to model GTX 980.

| 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 |

TABLE 4: SCU hardware parameters.

| Component | Requirements |
|---|---|
| Frequency | 1.27GHz |
| Technology | 32 nm |
| Pipeline Width | 4 elements/cycle |
| Vector Buffering | 5 KB |
| FIFO Requests Buffer | 38 KB |
| Hash Requests Buffer | 18 KB |
| Coalescing Unit | 32 assoc, 4-merge |

TABLE 5: IRU hardware requirements per partition.

| Component | Requirements |
|---|---|
| Requests Buffer | 2 KB |
| Classifier Buffer | 1.2 KB |
| Ring Buffer | 2.8 KB |
| Hash Data | 80 KB |

of main memory allowing us to evaluate a GDDR5 configured with 4 channels and 224 GB/s of bandwidth.

On the other hand, we have implemented the IRU architecture in GPGPU-Sim. To properly integrate the IRU into the GPGPU-Sim simulator the decoding is modified to extend the ISA. We also include small modifications to the LD/ST unit to handle the IRU instructions. The IRU is distributed among the memory partitions of the GPU. Each partition of the IRU uses a 2 KB FIFO to buffer requests. A buffer of 1.2 KB is used 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 table, which is a direct mapping hash table with 1024 sets, split in 4 physical partitions. Each IRU partition consists of two banks that store 256 sets in total, which represent 80 KB of on-chip storage, significantly smaller than the 512 KB of the L2 partition. Table 5 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 [20] with a node technology of 32 nm.

Finally, to evaluate our proposal we use state-of-the-art GPGPU implementations of BFS [10], SSSP [16], and PageRank [4] graph algorithms evaluated with benchmarks datasets in Table 2, collected from well-known repositories of research graph datasets [21], [22]. These graphs are representative of different application domains with varied sizes, characteristics and degrees of connectivity.

## 6 Experimental Results

In this section, we evaluate the improvements in performance and energy consumption achieved by our ISCU scheme. We evaluate four different configurations. Configuration *GPU* represents a pure software CUDA implementation of the graph algorithms running on an NVIDIA GTX 980. Configuration *SCU* is the system presented in [8] that combines the GPU and the SCU. The system *IRU* is the GPU extended with the IRU hardware as described in Section 2. Finally, configuration *ISCU* is our scheme as described in Section 3.

We first evaluate the energy consumption and performance of the ISCU in sections 6.1 and 6.2 respectively, using as the baseline the NVIDIA GTX 980. Afterwards, we compare the performance and energy of the ISCU with the SCU and the IRU in Section 6.3. Finally, we analyze the memory improvements of the ISCU in Section 6.4 and discuss its area requirements in Section 6.5.

### 6.1 Energy Evaluation

Offloading compaction operations to our ISCU provides consistent and large energy savings. Figure 15 shows the normalized energy consumption achieved by the ISCU over the baseline GPU system for all graphs and datasets. Additionally, the figure indicates the source of the remaining energy consumption distinguishing between the GPU, the majority, and the ISCU. On average, the ISCU delivers a reduction of 90% in energy consumption, achieving an energy reduction of 92%, 91% and 85% for BFS, SSSP and PR respectively. Several sources contribute towards energy savings. First, stream compaction offloading to the ISCU efficient hardware reduces static and dynamic energy consumption required to perform compaction operations. Second, workload filtering and memory coalescing provided by the ISCU improves GPGPU resources utilization, which lowers overall GPGPU energy consumption. Third, irregular access optimization enabled by the IRU further reduces memory contention. Finally, performance speedup further reduces static energy consumption of the system. Note that graph datasets with higher inter-connectivity see more energy savings due to higher computation offloading and increased workload filtering.

### 6.2 Performance Evaluation

The ISCU delivers significant speedups across different graph algorithms and datasets as seen in Figure 16, which shows

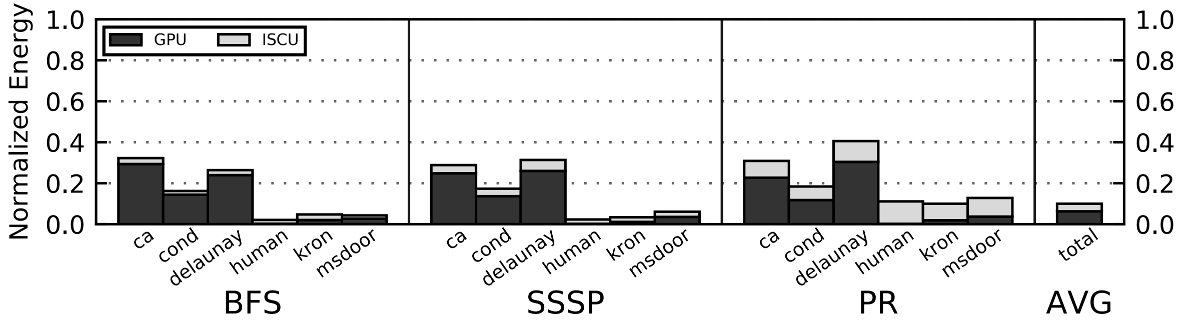


Fig. 15: Normalized energy consumption of the ISCU-enabled GPU compared to the GPU system (GTX 980), showing the split between GPU and ISCU energy consumption. Significant energy savings achieved across BFS, SSSP and PR graph algorithms and every dataset.

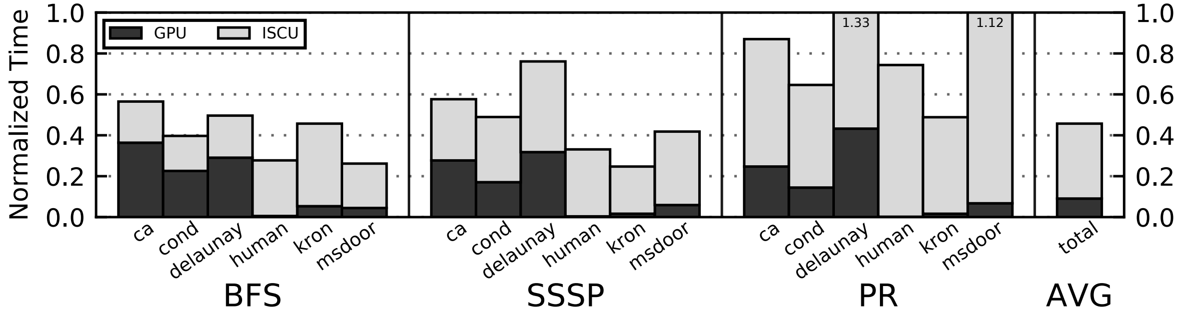


Fig. 16: Normalized execution time of the ISCU enabled GPU compared to the baseline GPU system, showing the split between GPU and ISCU execution time. Significant speedups are achieved across BFS, SSSP and PR graph algorithms and the majority of the datasets.

normalized execution time using the ISCU over the baseline GPU system. Additionally, the figure indicates the split of the execution time between the GPU and the ISCU, highlighting the high performance improvements over GPU execution of the higher inter-connectivity graphs. On average, the ISCU achieves a speedup of 2.2x with average speedups of 2.8x, 2.56x and 1.44x for BFS, SSSP and PR respectively. The efficiency of the ISCU is not as high for PR which in some cases incurs in overheads, a consequence of the large frontiers due to PR exploring the entire graph at every iteration. For PR, since every element is accessed on each iteration, all the data in the graph dataset is accessed incurring in less sparse accesses and higher locality. Nonetheless, the overheads observed are compensated by the high reduction in energy achieved due to the offloading of the compaction operations.

Overall, performance improvements are obtained from several sources. First, the efficient execution on hardware tailor-made for stream compaction operations delivers better performance than GPU architectures. Second, the ISCU pre-processing reduces GPGPU workload, additionally reducing GPGPU atomic synchronization overheads and improving memory coalescing. Third, the irregular accesses optimization enabled by the IRU further improves GPGPU performance by increasing memory coalescing.

### 6.3 Comparison with SCU and IRU

We compare the energy savings and speedups achieved with the ISCU against previous GPGPU architectural extensions for graph processing. Figure 17 shows how by combining in the ISCU the strengths of the SCU and IRU we are able to achieve a synergistic energy improvement, reaching on average a huge 10x improvement

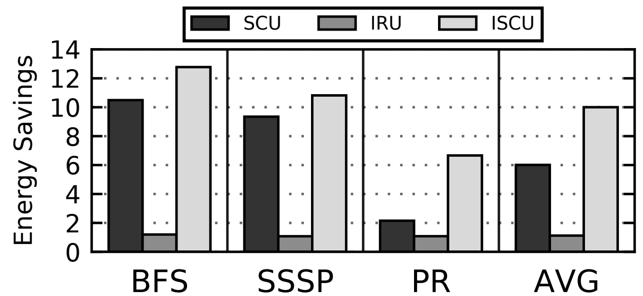


Fig. 17: Energy savings of the SCU, IRU and ISCU with respect to the baseline GPU system. The ISCU synergetically improves energy savings achieved with SCU and IRU.

in energy consumption compared to the GPU baseline, even though the SCU and IRU achieved on average 6x and 1.13x respectively. The big factor contributing to energy savings is delegating stream compaction operations to our specialized compaction hardware, as such the IRU optimizations do not deliver such huge energy savings. Note that the less sparse exploration performed by PR reduces its the energy savings. Furthermore, the ISCU avoids a large percentage of NoC transactions compared to the SCU, as the filtering is performed directly in the memory partitions.

We obtain synergistic performance improvements as seen in Figure 18, where the ISCU achieves on average a important 2.2x speedup compared to the baseline GPU, while the SCU and IRU

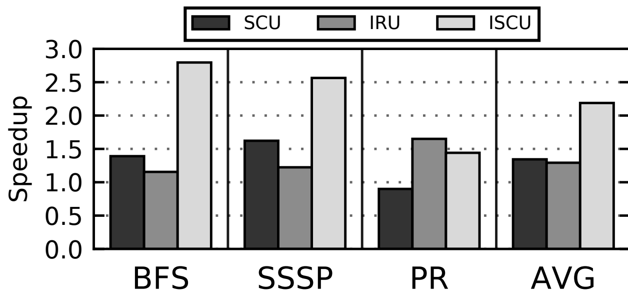


Fig. 18: Speedup of the SCU, IRU and ISCU with respect to the Baseline GPU system. The ISCU synergetically improves speedups achieved with SCU and IRU.

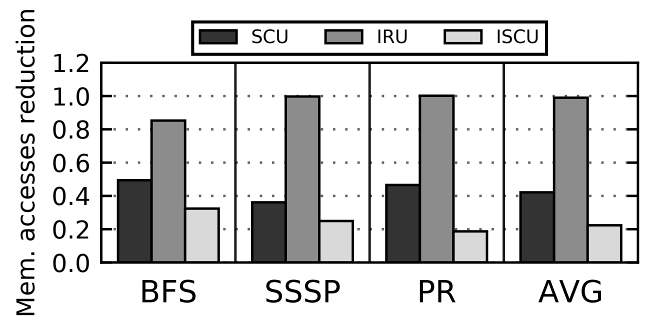


Fig. 19: Normalized memory accesses of the SCU, IRU and ISCU with respect to the baseline GPU system.

deliver on average 1.37x and 1.33x speedups respectively. The ISCU manages to overcome the overheads that impact PR filtering enhanced SCU instrumentation, which enable higher performance and energy savings. Although the IRU achieves a better speedup for PR than the ISCU, the minimal performance difference is more than made up by the significant difference in energy efficiency reaching 6.66x for ISCU against a 1.08x for the IRU.

### 6.4 Memory Improvements Evaluation

The ISCU significantly contributes to reduce memory accesses performed by graph processing applications as seen in Figure 19. The ISCU achieves on average a reduction of 78% in the total memory accesses performed by the baseline GPU, while the SCU and IRU achieve a reduction of 58% and 1% respectively. Although IRU memory accesses reduction is low, it significantly contributes to reduce intra-GPU memory resource utilization and interconnection traffic.

### 6.5 Area Overhead Evaluation

We evaluate the overhead of the complete ISCU which contains the improved SCU and the IRU hardware extensions. We do so by synthesizing and characterizing the different components, which require a total of 37.17 $mm^2$ additional area for the system. Considering the overall GPU system, the ISCU represents a 8.5% of the total GPU area. The ISCU area overhead is very low given the high energy savings and speedups achieved. In terms of both performance/area and energy, the ISCU results in very high benefits compared with the baseline and the SCU and IRU solutions.

## 7 Related Work

GPGPU graph-based applications face many challenges that steam from sparse and irregular memory access patterns and high memory divergence. Nonetheless, due to GPGPU architecture high-throughput many works have explored graph processing.

Software optimization approaches have explored branch divergence load balancing [10], [25], improved stream compaction implementations [11] and overall data structure optimizations [26], [27]. Many of these approaches incur in costly software optimizations or significant programming effort to overhaul an application. In contrast, our solution provides efficient optimizations with light amenable modifications improving overall GPGPU efficiency.

Graph framework approaches have been thoroughly explored with works such as Gunrock [12] implementing data-centric abstraction centered on operations on a node or edge frontier, HPGA [14] sparse matrix mappings, MapGraph [15] dynamically scheduling strategies and NVIDIA nvGRAPH [13]. Efficient GPGPU graph processing is challenging as many implementations show significant under-utilization and inefficiencies that we improve with the ISCU.

Many works propose the entire replacement of the GPU with custom-made accelerators for graph processing, setting aside the GPU due to its irregular execution limitations and inefficiencies. Proposals include standalone approaches such as TuNao [28], Dram-based Graphicionado [29], PIM-based GraphH [30]. In contrast, our solution leverages the popularity of GPU architectures while providing architectural improvements ameliorating irregular graph processing shortcomings.

Finally, this work is based on two previous GPGPU architectural extensions for graph processing, the SCU [8] and the IRU. We provide a detailed quantitative comparison with these two previous proposals in Section 6, showing that the ISCU achieves significant improvements in performance and energy consumption.

## 8 Conclusions

In this paper, we propose the IRU-enhanced SCU (ISCU), a GPGPU hardware extension that efficiently performs stream compaction operations commonly used by graph-based applications. The ISCU combines the strengths of the SCU and IRU hardware extensions to synergistically achieve high performance and energy-efficiency for GPGPU graph-based applications.

The ISCU solves the bottlenecks caused by the in-memory hash table used in the SCU to filter duplicated elements, that requires a large amount of traffic in the NoC. We propose to leverage the efficient IRU hash mechanism to perform filtering operations in the memory partitions, saving NoC traffic by a large extent and achieving significant speedups and energy savings.

The ISCU optimizations for graph processing operations deliver on average a 2.2x speedup and a reduction of 90% in energy consumption for a diverse set of graph-based applications and datasets, while achieving a high reduction of 78% in memory accesses, at the expense of a 8.5% GPU area overhead.

## Acknowledgment

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

## References


[1] C. Root and T. Mostak, "Mapd: a gpu-powered big data analytics and visualization platform," in *ACM SIGGRAPH 2016 Talks*, 2016, pp. 1–2.

[2] 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.

[3] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, 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.

[4] 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.

[5] A. Segura Salvador, "Characterization of speech recognition systems on gpu architectures," Master's thesis, Universitat Politècnica de Catalunya, 2016.

[6] D. Reinsel, J. Gantz, and J. Rydning, "The digitization of the world from edge to core," *IDC White Paper*, 2018.

[7] 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.

[8] 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*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 424–435. [Online]. Available: https://doi.org/10.1145/3307650.3322254

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] NVIDIA. nvgraph. [Online]. Available: https://developer.nvidia.com/nvgraph

[14] H. Yang, H. Su, M. Wen, and C. Zhang, "Hpga: A high-performance graph analytics framework on the gpu," in *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*. IEEE, 2018, pp. 488–492.

[15] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*, 2014, pp. 1–6.

[16] 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.

[17] N. G. GTX, "980 whitepaper," *NVIDIA Corporation*, 2014.

[18] 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.

[19] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41. ACM, 2013, pp. 487–498.

[20] 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.

[21] 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.

[22] DIMACS. (2010) 10th dimacs implementation challenge - graph partitioning and graph clustering. [Online]. Available: https://www.cc.gatech.edu/dimacs10/

[23] D. Compiler, "Synopsys inc," 2000.

[24] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: a cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, 2020.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: 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.

[30] 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.

**Albert Segura** received his B.Sc. degree in Computer Engineering in 2014, and his M.Sc. degree in MIRI: High Performance Computing in 2016, both from Universitat Politècnica de Catalunya (UPC - BarcelonaTech). He joined the ARCO (ARchitecture and COmpilers) research group at UPC in September 2015 and he is currently pursuing a Ph.D. in Computer Architecture at the UPC. His research focuses on the area of Graph processing on GPGPU Architectures. Contact him at asegura@ac.upc.edu.

**Jose-Maria Arnau** received Ph.D. on Computer Architecture from the Universitat Politècnica de Catalunya (UPC) in 2015. He is a postdoctoral researcher at UPC BarcelonaTech and a member of the ARCO (ARchitecture and COmpilers) research group at UPC. His research interests include low-power architectures for cognitive computing, especially in the area of automatic speech recognition and object recognition. Contact him at jarnau@ac.upc.edu.

**Antonio González** (Ph.D. 1989) is a Full Professor at the Computer Architecture Department of the Universitat Politcnica de Catalunya, Barcelona (Spain), and the director of the Microarchitecture and Compiler research group. He was the founding director of the Intel Barcelona Research Center from 2002 to 2014. His research has focused on computer architecture, compilers and parallel processing, with a special emphasis on microarchitecture and code generation. He has published over 370 papers, and has served as associate editor of five IEEE and ACM journals, program chair for ISCA, MICRO, HPCA, ICS and ISPASS and general chair for MICRO and HPCA. He is an IEEE Fellow. Contact him at antonio@ac.upc.edu.