SCU: A GPU Stream Compaction Unit for Graph Processing

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

ABSTRACT

Graph processing algorithms are key in many emerging applications in areas such as machine learning and data analytics. Although the processing of large scale graphs exhibits a high degree of parallelism, the memory access pattern tend to be highly irregular, leading to poor GPGPU efficiency due to memory divergence. To ameliorate this issue, GPGPU applications perform a stream compaction operation each iteration of the algorithm to extract the subset of active nodes/edges, so subsequent steps work on compacted dataset.

We show that GPGPU architectures are inefficient for stream compaction, and propose to offload this task to a programmable Stream Compaction Unit (SCU) tailored to the requirements of this kernel. The SCU is a small unit tightly integrated in the GPU that efficiently gathers the active nodes/edges into a compacted array in memory. Applications can make use of it through a simple API. The remaining steps of the graph-based algorithm are executed on the GPU cores taking benefit of the large amount of parallelism in the GPU, but they operate on the SCU-prepared data and achieve larger memory coalescing and, hence, much higher efficiency. Besides, the SCU performs filtering of repeated and already visited nodes during the compaction process, significantly reducing GPGPU workload, and writes the compacted nodes/edges in an order that improves memory coalescing by reducing memory divergence.

We evaluate the performance of a state-of-the-art GPGPU architecture extended with our SCU for a wide variety of applications. Results show that for high-performance and for low-power GPU systems the SCU achieves speedups of 1.37x and 2.32x, 84.7% and 69% energy savings, and an area increase of 3.3% and 4.1% respectively.

CCS CONCEPTS

• **Computing methodologies** → **Graphics processors**; • **Hardware** → *Hardware accelerators*.

KEYWORDS

GPGPU, graph processing, stream compaction

ACM Reference Format:
Albert Segura, Jose-Maria Arnau, Antonio González. 2019. SCU: A GPU Stream Compaction Unit for Graph Processing. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322254

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322254

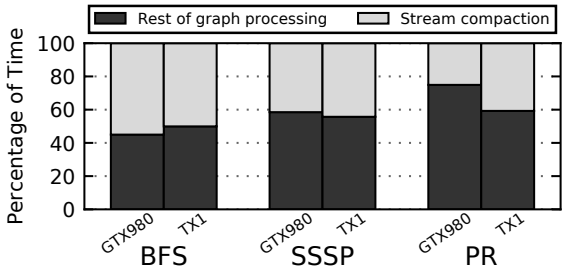


Figure 1: Breakdown of the average execution time for several applications and three graph primitives (BFS, SSSP and PR). Measured on an NVIDIA GTX 980 and Tegra X1.

1 INTRODUCTION

Graph processing algorithms are at the heart of popular machine learning [1–3] and data analytics [4–6] applications. Processing of large graphs exhibits a high degree of parallelism [7] which can potentially benefit from highly-parallel GPGPU architectures. However, data in graph-based applications are typically unstructured and irregular [7, 8], which results in sparse and unpredictable memory access patterns. In addition, graph processing shows extremely low computation to memory access ratio [9]. Not surprisingly, GPU-based graph processing suffers from low efficiency [8], since GPGPU architectures are optimized for compute-intensive workloads with regular memory access patterns.

To mitigate the aforementioned problems, GPU-based graph algorithms gather the required nodes/edges from a large graph into a compacted array. Typically, only a sparsely distributed subset of the nodes/edges of the graph are visited on every iteration of the graph algorithm. By compacting those elements into an array in consecutive memory locations, subsequent graph processing on the compacted dataset exhibits a more regular memory access pattern. This gathering is typically performed via a stream compaction operation [10]. Stream compaction is a common primitive used to filter out unwanted elements in sparse data, with the aim of improving the performance of parallel algorithms that work on the compacted dataset. In Figure 1 we show the average percentage of time spent on stream compaction for three commonly used graph kernels: Breadth First Search (BFS), Single Source Shortest Path (SSSP) and PageRank (PR). As it can be seen, stream compaction represents between 25% to 55% of the total execution time. Although state-of-the-art CUDA implementations of BFS [11] and SSSP [12] mix compaction and processing, we split them just for the purpose of making Figure 1, as our best effort to quantify the cost of compaction.

We claim that GPGPU architectures are not efficient for stream compaction workloads for several reasons. First, stream compaction consists of sparse memory accesses with poor locality that fetch the elements (nodes/edges) to be compacted. Sparse memory accesses result in very low memory coalescing, producing intra-warp memory divergence and reducing GPU efficiency by a large extent. Second, stream compaction has an extremely low computation to memory access ratio, as it primarily consists of load and store instructions to move data around in main memory. GPU cores, a.k.a. streaming multiprocessors, are optimized for compute-intensive workloads, including hundreds of functional units that are largely underutilized during the stream compaction stage.

Since GPU architectures are not designed to efficiently support compaction operations, we propose to extend the GPU with a novel unit tailored to the requirements of stream compaction. We term this hardware as the Stream Compaction Unit (SCU). The SCU requires a very small area, is tightly integrated in the GPU and is programmed through a simple API. By providing such integration and simple API, we offload compaction operations to the SCU to achieve higher performance and energy efficiency for this phase, and maximize the effectiveness of the streaming multiprocessors for the phases of the algorithm that work on the compacted dataset.

In addition to rearranging the data in memory, the SCU filters out duplicated elements while the data is compacted. This filtering results in a massive reduction in GPU workload, especially for large-scale graphs. Besides, the SCU writes the compacted data in an order that improves memory coalescing for the remaining GPU processing. The key idea is to store together edges whose destination nodes are in the same cache line. By doing so, the memory accesses for processing the compacted data can be better coalesced, since the different threads in a warp will issue memory requests to the same cache line. Although performing these additional operations increases the workload of the SCU, it provides a large reduction in the activity of the streaming multiprocessors, resulting in net savings in execution time and energy consumption for the overall system.

To summarize, this paper focuses on high performance and energy efficient graph processing on GPGPU architectures. Its main contributions are the following:

- We characterize the performance and energy consumption of the stream compaction operation on a modern GPU architecture, showing that it takes more than 50% of the execution time and more than 45% of the energy consumption for graph processing applications.
- We propose the SCU, a novel unit that is tailored to the requirements of stream compaction, and describe how this unit can be integrated in existing GPGPU architectures.
- We extend the SCU to perform filtering of duplicated nodes, which removes 75% of GPU workload on average, and to rearrange the compacted data to reduce memory divergence, which improves memory coalescing by 27%.
- Overall, the high-performance and low-power GPU designs including our SCU unit achieve speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively on average for several graph-based applications. The SCU represents a small area overhead of 3.3% and 4.1% respectively.

The rest of the paper is organized as follows. Section 2 reviews the state-of-the-art on graph processing in GPGPU systems. Section 3 presents the basic architecture of the SCU, whereas Section 4 describes the extensions for filtering duplicated nodes and improving memory coalescing. Section 5 presents the evaluation methodology and Section 6 provides the experimental results. Section 7 reviews some related work and, finally, Section 8 sums up the main conclusions of this work.

2 GRAPH PROCESSING ON GPGPUS

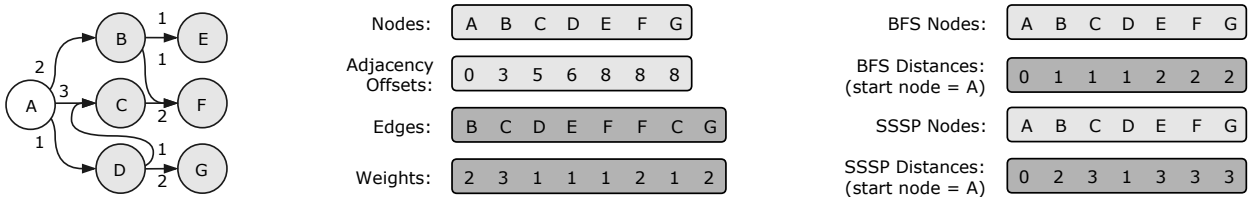
Many problems in data analytics, machine learning and other areas can be described and solved using graph algorithms. These algorithms employ graphs to describe the relationships on a given dataset of interest, and explore them to extract the desired information. GPGPU architectures can be used to accelerate graph processing by exploring in parallel multiple nodes and their outgoing edges. However, graph-based algorithms exhibit characteristics that are not amenable for GPGPU: low computation to memory access ratio, irregular memory access patterns and poor data locality [7].

To ameliorate these issues, CUDA/OpenCL implementations of graph algorithms leverage several solutions. First, graph applications employ compact and efficient representation of the graph data structure, Compressed Sparse Row (CSR) [13] being one of the most popular formats. Figure 2 shows a reference graph with its corresponding CSR representation. This format consists of an array with all the nodes in the graph, two arrays containing all the edges and their corresponding weights respectively, and an array with the adjacency offsets. Besides, GPGPU-based graph algorithms employ different approaches to avoid expanding duplicated nodes, typically by using atomic operations. For example, the parallel processing of nodes A and D in the graph of Figure 2a would generate two copies of the same node C in a graph traversal kernel. GPGPU implementations eliminate/reduce the amount of duplicated nodes by accurately/loosely tracking already visited nodes. Finally, to deal with sparse and irregular memory accesses, GPGPU graph processing leverages stream compaction techniques [14] to gather the sparse data in contiguous memory locations, improving memory coalescing. The compacted array of nodes/edges is typically referred as the *frontier*, which is a contiguous space in memory containing the nodes/edges that are being explored in a given iteration of the algorithm.

In this work we focus on Breadth-First Search (BFS) [11], Single-Source Shortest Paths (SSSP) [12] and PageRank (PR) [15] functions, that are among the most widely used primitives for graph processing.

2.1 Breadth-First Search

Breadth-First Search (BFS) is a graph primitive that computes the minimum distance, in terms of traversed edges, from a given node to all the nodes in a graph (see Figure 2c). We use the CUDA implementation of BFS proposed in [11], that includes several techniques to mitigate issues of GPU-based graph processing mentioned in Section 2. The exploration of a graph starts in a given node and an iterative process is done until all nodes are visited. Each iteration consists of an expansion phase and a contraction phase (see Figure 3).



(a) Reference Graph **(b) CSR Representation** **(c) Graph Exploration**

Figure 2: Graph example (a), corresponding CSR representation (b) and exploration results using BFS and SSSP (c).

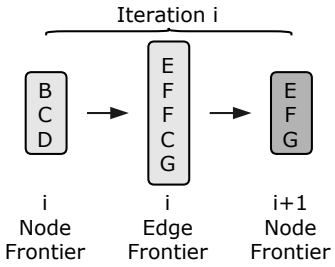


Figure 3: Node and edge frontier resulting from an iteration of a BFS exploration of graph 2a.

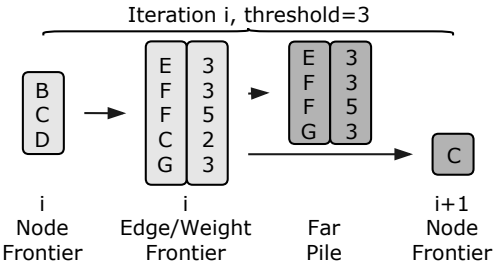


Figure 4: Data structures resulting from an iteration of the SSSP algorithm of graph 2a.

2.1.1 ***Expansion phase:*** It consumes the node frontier and generates the new edge frontier. The main challenge is workload balancing, since different nodes exhibit largely different number of outgoing edges. Several mechanisms are used in [11] to cooperatively expand the edges of a node or several nodes, increasing the warp-level synchronizations but improving workload balance by a large extent.

2.1.2 ***Contraction phase:*** It consumes the edge frontier to create the new node frontier, updating the information of different nodes during this process. The main challenge is detecting already visited nodes. A state-of-the-art solution proposed in [11] is to use a "best-effort" bitmask, i.e. updated without using atomic operations, which may yield false negatives due to race conditions but removes overheads of atomic operations.

2.2 Single-Source Shortest Paths

Single-Source Shortest Paths (SSSP) computes the minimum cost from a source node to all the nodes in a graph, where the cost is the addition of the weights of the traversed edges (see Figure 2c). We use the CUDA implementation presented in [12], which is similar to BFS. However, the edge frontier is split into the "near" and "far" frontiers for paths with low and high cost respectively, based on a dynamically adjusted threshold. The most promising paths in the "near" frontier, with the lower cost of traversal, are expanded first in order to reduce the cost of revisiting and updating nodes in following iterations of the algorithm.

On each iteration, the expansion phase consumes the node frontier to generate the edge and weight frontiers. Next, the contraction phase stores high cost nodes into the "far" pile (see Figure 4), whereas low cost nodes are used to create the new node frontier. This process is repeated until the node frontier becomes empty. At this point, the threshold is updated and the contract phase starts consuming nodes from the "far" pile.

2.2.1 ***Expansion phase:*** It generates the new edge and weight frontiers, using the same mechanisms for workload balancing as described in Section 2.1.1.

2.2.2 ***Contraction phases:*** The main challenge is the detection and filtering of duplicated edges. The implementation proposed in [12] employs a lookup table with one entry per node in the graph. Each thread writes its ID to the corresponding lookup table entry and, after synchronization, only threads whose ID is stored in the lookup table are allowed to modify the new node frontier. Although no atomic operation is used to access the lookup table, the cost of the nodes (SSSP Distances in Figure 2c) is updated with atomicMin to guarantee it is the shortest path.

2.3 PageRank

PageRank (PR) is a well-known graph primitive used in search engines [16]. PR computes the scores of the nodes based on the following equation to iteratively update all nodes until the algorithm converges, where α is a constant (dampening factor) and U_{deg} is the out degree of node U:

$$V_{score} = \alpha + 1 - \alpha \frac{U_{score}}{U_{deg}}$$

We borrow the CUDA implementation of PR proposed in [15]. Each iteration of the algorithm consists of four phases: expansion, rank update, dampening and convergence check.

2.3.1 ***Expansion phase:*** It generates the edge frontier and weight (i.e. rank) frontier, using several workload balance mechanisms (see Section 2.1.1). The rank of each node is divided by its out degree.

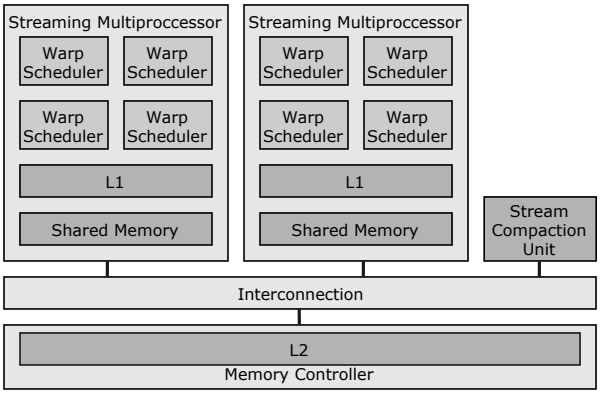


Figure 5: Tegra X1 GPGPU architecture including the SCU.

2.3.2 ***Rank Update phase:*** It computes the new ranks using atomic addition operations. An atomic operation is issued for every edge in the graph since, unlike BFS or SSSP, all the nodes are considered active on every iteration of the algorithm.

2.3.3 ***Dampening phase:*** The dampening factor is applied to the rank of each node. This phase is well suited for execution on the GPU.

2.3.4 ***Convergence check phase:*** The ranks for the current iteration are compared with the ranks from the previous iteration, and the algorithm finishes if the maximum node-wise difference is smaller than a given epsilon value. As the dampening phase, convergence check phase is GPU-friendly.

3 STREAM COMPACTION UNIT

In this section, we propose a Stream Compaction Unit (SCU) tailored to the requirements of graph applications. Compacting the active nodes/edges in consecutive memory locations is key for achieving high utilization of GPU resources. However, the GPU is inefficient performing stream compaction operations, as it only requires data movements with no computation on the data, but GPU architectures are optimized for compute intensive workloads. Furthermore, the memory accesses for compacting the data are typically sparse and highly irregular, leading to poor memory coalescing. As shown in Figure 1, the stream compaction operation takes more than 40% of GPU time in several graph applications.

We propose to offload the stream compaction operations to a specialized unit, the SCU. The SCU is an efficient, compact and small footprint unit that is attached to the streaming multiprocessor interconnection network as shown in Figure 5. The SCU performs data compaction in a sequential manner, avoiding synchronization and work distribution overheads, and operates with just the minimum hardware requirements to perform data movement operations for stream compaction workloads.

Our proposed graph processing approach exploits the parallelism of the GPU to explore a graph while making use of the SCU to perform the data compaction operations. Once the compaction phase of the algorithm starts, SCU operations are issued, and the data compaction is performed on the sparse data in memory and compacted into a destination array in memory. Once the operation concludes, the compacted data is available to the GPU which resumes execution continuing the graph exploration.

3.1 SCU Compaction Operations

The SCU is a programmable unit which includes a number of generic data compaction operations that allow a complete implementation of stream compaction. Figure 6 shows the operations operations supported by the SCU and interact with data. All SCU operations have some parameters which are omitted from the figure for the sake of simplicity: the size of the data and the number of elements on which to operate. The SCU 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 set to 1 if the element in the input data evaluates to true using the comparison operator and the reference value, and to 0 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 using the bitmask vector. The output at the destination contains only valid elements preserving the original order.
- **Replication Compaction:** An 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 corresponding entry in the count vector.

3.2 Hardware Pipeline

The SCU implements the operations previously described with the hardware illustrated in Figure 7. The SCU consists of five different functional units. An operation begins by configuring the main component, the Address Generator.

3.2.1 ***Address Generator:*** It is configured at the beginning of each operation with the corresponding parameters, and begins execution of the operation by generating the addresses of the data to be compacted, as well as, if needed, the other vector parameters: bitmask, indexes and count. The generated addresses are stored on a small buffer to avoid potential stalls.

3.2.2 ***Data Fetch:*** This is a straightforward component which generates data memory requests to the addresses generated by the Address Generation component. The requests are sent to the Coalescing

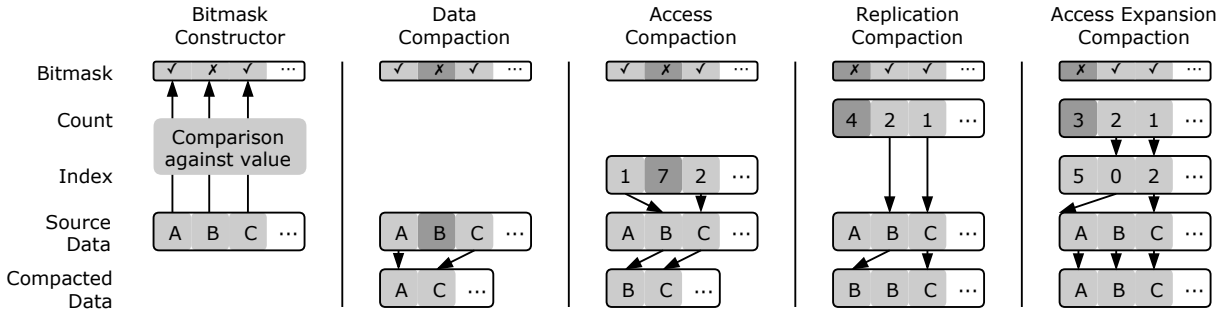


Figure 6: SCU operations required to implement stream compaction capabilities, illustrated with the data that each operation uses and generates. Arrow direction indicates flow of data.

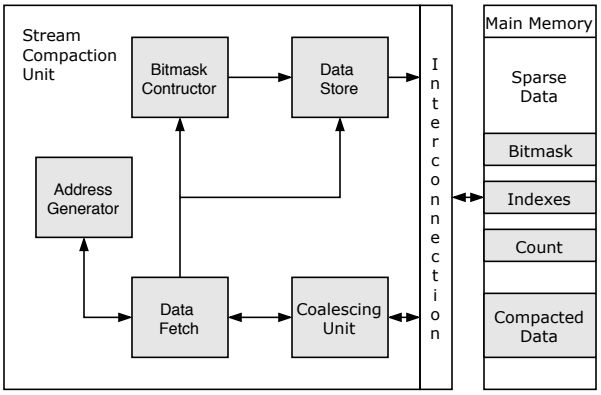


Figure 7: Architecture of the Stream Compaction Unit.

Unit and the order of requests is preserved. When the data is received, it is forwarded to either the Bitmask Constructor or the Data Store component, respecting the FIFO order. For this purpose, it uses a FIFO queue that stores the data until it can be forwarded to the consumer component.

3.2.3 ***Coalescing Unit:*** This unit coalesces read memory requests to the same cache memory block in order to reduce congestion of the interconnection and memory requests to upper levels of the memory hierarchy, which reduces energy consumption.

3.2.4 ***Bitmask Constructor:*** It is the specialized component used by the bitmask constructor operation to generate the bitmask vector that is used by other operations. Contains logic to perform comparison operations of a reference value against each of the elements that it receives.

3.2.5 ***Data Store:*** It is the component that receives the data generated by the other components and generates the consequent write memory requests. Since data is stored in consecutive memory addresses, this unit includes a simple coalescing of write operations to minimize requests and interconnection network traffic.

3.3 Breadth-First Search with the SCU

Both phases of BFS, expansion and contraction, include compaction operations that can be offloaded to the SCU. Algorithm 1 summarizes the modifications.

3.3.1 ***Expansion phase:*** The main workload of this phase is offloaded to the SCU using the Access Expansion Compaction operation. The indexes and count vectors are efficiently prepared by the GPU as it requires contiguous memory accesses. Each entry in the indexes vector represents the offset in the edges vector where the first edge of the corresponding node is stored. The count vector stores for a node its number of edges.

3.3.2 ***Contraction phase:*** This phase takes as an input the edge frontier generated by the expansion phase and generates the new node frontier. It filters out duplicated edges and already visited nodes. The GPU is responsible for generating the mask that will be used for the filtering. Then, the compaction of valid nodes is offloaded to the SCU, by using Data Compaction operation that has the edge frontier and the bitmask vector as inputs.

Algorithm 1 BFS using the SCU.
nf: nodeFrontier, **ef:** edgeFrontier

```
procedure EXPANSION(nf)
    count, indexes ← preparationGPU(nf)
    ef ← accessExpansionCompactionSCU(
            edges, count, indexes)
    return ef
end procedure
procedure CONTRACTION(ef)
    bitmask ← contractionGPU(ef)
    nf ← dataCompactionSCU(ef, bitmask)
    return nf
end procedure
```

3.4 Single-Source Shortest Paths with the SCU

Similar to BFS, both expansion and the two contraction phases of the algorithm include stream compaction operations that can be offloaded to the SCU. Algorithm 2 summarizes the modifications.

3.4.1 ***Expansion phase:*** The GPU generates the indexes and count vectors, then the SCU generates the edge frontier. Next, the SCU has to generate the weights vector corresponding to the edge frontier. This vector contains the costs associated with the edges which are obtained using two operations: an Access Expansion Compaction and a Replication Compaction operation. The former operation generates the weights associated to each edge, and the latter adds its accumulated cost.

3.4.2 ***Contraction phases:*** This phase takes as an input the edge frontier and filters out duplicated edges and already visited nodes, updates the node's information, and compacts valid nodes of the next frontier. Edges with accumulated cost higher than the current iteration threshold are pushed to the back of the "far" pile. The operations that are offloaded to the SCU are the compaction of valid nodes and the compaction of high cost edges in the "far" pile. In both cases, a Data Compaction operation is used. The GPU is responsible for computing the bitmask, both for low-cost and high-cost edges (near and far respectively) that the SCU will use for filtering.

Algorithm 2 SSSP using the SCU.
nf: nodeFrontier, **ef:** edgeFrontier, **wf:** weightFrontier

```
procedure EXPANSION(nf)
    indexes, count ← preparationGPU(nf)
    ef ← accessExpansionCompactionSCU(
            edges, indexes, count)
    wf ← accessExpansionCompactionSCU(
            weights, indexes, count)
    wf ← replicationCompactionSCU(weights, count)
    return ef, wf
end procedure
procedure CONTRACTION(ef, wf, threshold)
    bitmaskNear, bitmaskFar ← contractionGPU(
            ef, wf, threshold)
    nf ← dataCompactionSCU(ef, bitmaskNear)
    farPileEdges ← dataCompactionSCU(
            ef, bitmaskFar)
    farPileWeights ← dataCompactionSCU(
            wf, bitmaskFar)
    return nf
end procedure
```

3.5 PageRank with the SCU

Only the expansion phase of PR performs stream compaction operations. PR does not operate with a node frontier, since it explores all the nodes and edges on every iteration of the algorithm. Algorithm 3 summarizes the modifications.

3.5.1 ***Expansion phase:*** The GPU creates the indexes, count and weight vectors. Afterwards the SCU generates the edge frontier, in the same way as it is done for BFS. Finally, the weight frontier is generated using the pre-processed weight vector that contains the original weight of each node, but divided by the output degree of the node, so that the SCU can replicate this value for each edge with a Replication Compaction operation.

Algorithm 3 PR using the SCU.
ef: edgeFrontier, **wf:** weightFrontier

```
procedure EXPANSION(nodes)
    count, indexes, weights ← preparationGPU(nodes)
    ef ← accessExpansionCompactionSCU(
            edges, indexes, count)
    wf ← replicationCompactionSCU(weights, count)
    return ef, wf
end procedure
```

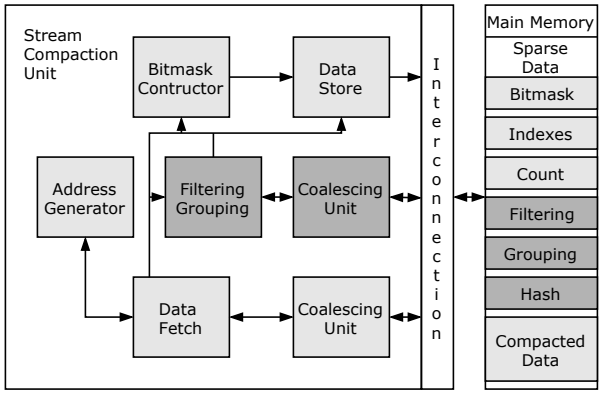


Figure 8: Improved architecture of the SCU adding filtering and grouping hardware and an additional coalescing unit.

4 FILTERING AND GROUPING

The Stream Compaction Unit (SCU) presented in Section 3 efficiently performs data compaction operations, while the remaining graph processing is performed by the GPU. The GPU graph processing has an important overhead due to nodes/edges duplication. Duplicated elements are a byproduct of parallel graph exploration on GPU architectures. Removing duplicates in the GPU requires costly mechanisms such as large lookup tables or atomic operations. Furthermore, GPU processing is very sensitive to the effectiveness of memory coalescing.

We propose to improve the SCU with the capability of further processing and delivering the compacted data in a more GPU-friendly manner. More specifically, we extend the SCU to remove duplicates and reorder compacted data elements to make them more effective for memory coalescing. We refer to this reorder step as grouping, since it is not a complete order but an order that tries to maximize the effectiveness of memory coalescing by grouping, i.e. storing together, edges whose destination nodes are in the same cache line. To this end, we incorporate a Filtering/Grouping unit and an additional coalescing unit as depicted in Figure 8.

4.1 Filtering/Grouping Unit

The Filtering/Grouping unit operates by storing each graph element (i.e. edge or node) into a hash table resident in main memory and cached in the L2. By placing the hash in memory we are able to reconfigure the hash organization to target Filtering or Grouping operations, since the requirements for both may be different. In addition, using existing memory does not require any additional

hardware. 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. To simplify the implementation, in case of collisions the corresponding hash table entry is overwritten. This means that false negatives are possible, but it largely simplifies the cost of the implementation while removing most of the duplicates with relatively small hash table sizes as shown in Section 6. For the grouping operation, the hash table is used to create groups of edges whose destination node lies in the same cache line, in order to store them together in the compacted array.

The filtering and grouping of compacted data is done by the SCU in a two step process. In the first step, the SCU performs a compaction operation and identifies the duplicated elements (i.e. generates a bitmask vector indicating the filtered elements) and the reordering required for the grouping (i.e. a reordering vector with indexes indicating the new order of the data). In the second step, the SCU uses the generated data to perform filtering and grouping on the compacted data. All compaction operations shown in Figure 6 can generate and operate with filtering and grouping data.

As an example, the compaction operation could be performing the expansion of the node frontier to generate the new edge frontier. The first step creates filtering and reordering information for each of the elements on the edge frontier. Note that the first operation does not generate the new edge frontier but the filtering or grouping information instead. Next, the second step employs the previously generated information and creates the edge frontier compacted, reordered and without the duplicated elements.

4.2 Filtering Operation

We use two filtering schemes: filtering by unique element (useful for BFS), filtering by unique-best cost (useful for SSSP). The hash table is configured with 4 bytes per block for BFS and or 8 bytes per block for SSSP. Further details are listed in Table 1.

The filtering operation generates a bitmask vector which contains a bit for every output data element. Each bit is set to one if the element is to be kept, or zero otherwise. The filtering operation works as follows. For each element (node or edge) to be compacted, the SCU computes its hash table entry by applying a hash function to its ID. If the corresponding hash table entry is empty, the element ID is stored in the hash table and the corresponding bitmask entry is set to one. If the same element ID is found in the hash table entry a duplicated node/edge is detected and, hence, the element is discarded (bitmask entry for that element is set to zero). In case a different element ID is found, the older element is evicted and the new element ID is stored in the hash table. Note that since we overwrite some elements in case of collisions the removal of duplicates is not complete. However, this implementation provides a good trade-off between complexity and effectiveness in removing duplicates.

The above process describes the filtering scheme of unique elements. In the case of unique-best cost filtering, an additional cost value is stored in the hash table. On a hit, further processing is done: if the element has a better cost, it overwrites the cost in the hash table entry.

4.3 Grouping Operation

Grouping is achieved using the same hash table with a different configuration. We use a hash table entry to store a number of elements that access the same memory block, which in our system shown in Table 1 can hold up to 32 elements of 4 bytes (line size of L2 cache). In our hashing scheme, however, each hash table entry is limited to grouping 8 elements of 4 bytes (i.e. it creates groups of 8 elements at most). We have experimentally observed that it is better to reduce the number of elements per group to 8, since more elements would imply to reduce the number of sets kept in the hash table for the same total capacity. Furthermore, in sparse datasets, increasing the number of elements per group to 32 provides negligible benefits even if storage was unbounded, as it is quite hard to fill them up.

The output of the grouping operation is a vector that indicates for each input element which order (i.e. position) it will occupy in the compacted array. The grouping process works as follows. For each element, the SCU computes the memory block (i.e. cache line) of the node/edge being processed. Next, a hash function is applied to the memory block number to group together elements that require the same memory block, with the aim of improving memory coalescing on the GPU. If the corresponding memory block is found in the hash table, the new element is added to the hash table entry. If the entry is occupied by a different memory block, the older block is evicted and the elements it contains are written in the output vector to guarantee that they will be stored together in the compacted array. Again, this scheme does not guarantee that all the elements that require the same memory block are stored together, but it is highly effective in practice while being amenable for hardware implementation.

4.4 Breadth-First Search with the Enhanced SCU

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, negating the performance benefits of the increased coalescing. Filtering reduces the GPU workload, in terms of edges and nodes, to 14% of the original workload on average. Shown on Algorithm 4, the required changes are the following:

4.4.1 ***Expansion phase:*** Requires an additional Access Expansion Compaction operation to construct the filtering vector. This vector is used by the following Access Expansion Compaction which generates the final filtered edge frontier.

4.4.2 ***Contraction phase:*** Requires an additional Data Compaction operation to construct the filtering vector. This filtering vector is used by the following Data Compaction operation which generates the final filtered node frontier. Note that this filtering is applied because the filtering done by BFS is not complete (as in SSSP). Otherwise, the filtering of the edge frontier would be done on the GPU when doing the graph exploration and further filtering at the SCU would not provide any benefit.

4.5 Single-Source Shortest Paths with the Enhanced SCU

Filtering out of duplicated elements is beneficial for both the expansion and contraction phases of the SSSP algorithm. Additionally,

Algorithm 4 Additional operations for BFS on the enhanced SCU.
nf: nodeFrontier, **ef:** edgeFrontier

```
procedure EXPANSION(nf)
    ... previous GPU operations ...
    filtering ← accessExpansionCompactionSCU(
            edges, indexes, count)
    ... previous SCU operations ...
    return ef
end procedure
procedure CONTRACTION(ef)
    ... previous GPU operations ...
    filtering ← dataCompactionSCU(ef, bitmask)
    ... previous SCU operations ...
    return nf
end procedure
```

Algorithm 5 Additional operations for SSSP on the enhanced SCU.
nf: nodeFrontier, **ef:** edgeFrontier, **wf:** weightFrontier

```
procedure EXPANSION(nf)
    ... previous GPU operations ...
    filtering ← accessExpansionCompactionSCU(
            edges, indexes, count)
    grouping ← accessExpansionCompactionSCU(
            edges, indexes, count)
    ... previous SCU operations ...
    return ef, wf
end procedure
procedure CONTRACTION(ef, wf, threshold)
    ... previous GPU operations ...
    grouping ← dataCompactionSCU(ef, bitmaskNear)
    ... previous SCU operations ...
    return nf
end procedure
```

unlike BFS, the grouping does not interfere with the GPU filtering, and the coalescing improvement results in a net gain in performance. The SCU reduces the GPU workload (i.e. nodes and edges) to 22% of the original workload on average, and improves the coalescing effectiveness by a factor of 27%. Shown on Algorithm 5, the required changes are the following:

4.5.1 ***Expansion phase:*** Two additional Access Expansion Compaction are required. One operation is responsible for constructing 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.

4.5.2 ***Contraction phases:*** 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 SCU filtering would result in no benefit. The grouping information is only used by the subsequent operation that operates on "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 operates on "far" elements and generate the new filtered and grouped node frontier.

4.6 PageRank with the Enhanced SCU

Removing duplicated or already visited nodes is not an option for PR, since it considers all the nodes on every iteration of the algorithm. Furthermore, since the application accesses the entire set of edges and nodes the memory access pattern is less irregular, hence, grouping the nodes provides little memory coalescing improvement. Therefore, the enhanced functionalities of the SCU are not used when running PR.

5 EVALUATION METHODOLOGY

We have developed a cycle-accurate simulator that models the architecture of our SCU. We match the SCU frequency to the one of the target GPU, being 1.27GHz and 1 GHz for GTX980 and TX1 respectively. 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. The filtering and grouping operations use a reconfigurable in-memory hash table, in addition to a 18 KB request buffer. Finally, the coalescing units hold up to 32 in-flights requests with a merge window of 4 elements. Table 1 shows the hardware configuration of the SCU.

Table 1: SCU hardware parameters

Technology, Frequency	32 nm, 1.27GHz / 1GHz
Vector Buffering	5 KB
FIFO Requests Buffer	38 KB
Hash Request Buffer	18 KB
Coalescing Unit	32 in-flight requests, 4-merge

We implemented the SCU design in Verilog, to obtain area and energy consumption we synthesized it using the Synopsis Design Compiler [17] and the technology library of 32nm from Synopsys with low power configured at 0.78V. We use CACTI [18] to characterize the cache and interconnection.

In addition, we have integrated our simulator with DramSim2 [19] to properly model main memory accesses, and we modified it to simulate a 4GB GDDR5 and a 4GB LPDDR4. We use GPUWattch [20] to obtain memory power measurements for GDDR5, and we use the Micron power model for LPDDR4 [21]. To evaluate the GPU performance we use GPGPU-Sim [22] configured to model both our target GPU systems. We model an NVIDIA GTX 980 with 4GB GDDR5 using the parameters shown in Table 3 and an NVIDIA Tegra X1 with 4GB LPDDR4, with the parameters shown in Table 4. We also use GPUWattch [20] to obtain GPU power and area measurements.

Table 5 shows the benchmark datasets we selected, which have been collected from a number of well known repositories [23, 24]

Table 2: SCU scalability parameters selection for the GTX980 and TX1 GPU

	GTX980	TX1
Pipeline Width	4 elements/cycle	1 elements/cycle
Filtering BFS Hash	1 MB, 16-way, 4 bytes/line	132 KB, 16-way, 4 bytes/line
Filtering SSSP Hash	1.5 MB, 16-way, 8 bytes/line	192 KB, 16-way, 8 bytes/line
Grouping SSSP Hash	1.2 MB, 16-way, 32 bytes/line	144 KB, 16-way, 32 bytes/line

Table 3: GPGPU-Sim high-performance GTX980 parameters

GPU, Frequency	NVIDIA GTX 980, 1.27GHz
Streaming Multiprocessors	16 (2048 threads), Maxwell
L1, L2 caches	32 KB, 2 MB
Shared Memory	64 KB
Main Memory	4 GB GDDR5, 224 GB/s

Table 4: GPGPU-Sim low-power Tegra X1 parameters

GPU, Frequency	NVIDIA Tegra X1, 1GHz
Streaming Multiprocessors	2 (256 threads), Maxwell
L1, L2 caches	32 KB, 256 KB
Shared Memory	64 KB
Main Memory	4 GB LPDDR4, 25.6 GB/s

and are representative of different categories of graphs as well as dimensions and connectivity properties.

5.1 SCU Scalability

High-performance GPUs are optimized for performance at the expense of power dissipation, whereas low-power GPUs provide more modest performance while keeping energy consumption extremely low. Due to this large variability in performance, a fixed design of the SCU will be undersized or oversized in some cases. Hence, a mechanism to scale the SCU is required, in order to adjust its performance, area and energy consumption to the requirements of the target segment. To this end, we provide two configuration parameters. The first one is the pipeline width of the SCU, i.e. the number of elements (nodes/edges) that can be processed per cycle. This parameter is included in the RTL code and the user can set an appropriate value before synthesizing the SCU. We found that a pipeline width of 1 provides a good trade-off between area and performance for the low-power TX1 GPU, whereas a pipeline width of 4 is required to outperform a high-performance GPU such as the GTX980. The second parameter to achieve scalability is size of the hash tables employed for filtering and grouping operations. Larger sizes potentially provide a more effective filtering and grouping, but may have a negative impact on performance if the L2 cache is too small. This parameter can be set by the user at runtime. Table 2 shows the parameters used for the evaluation on each GPU system.

6 EXPERIMENTAL RESULTS

In this section, we evaluate the performance improvement and energy reduction of the SCU. Figure 9 shows normalized energy consumption for BFS, SSSP and PR primitive on several graphs on our high-performance (GTX980) and low-power (TX1) GPU systems enhanced with the SCU, whereas Figure 10 shows the normalized execution time. The baseline configuration for both is the respective GPU system without our SCU.

6.1 Energy Evaluation

Figure 9 shows that the SCU provides consistent energy reduction, including both dynamic and static energy, across all graphs, all GPU systems and all graphs primitives. On average, the SCU provides a reduction in energy consumption of 6.55x for GTX980 and 3.24x for TX1 (an 84.7% and 69% of energy reduction respectively). The GTX980 is optimized for high performance at the expense of increased power dissipation, whereas the TX1 keeps energy consumption lower. For this reason, the energy savings is more significant on the GTX980, where we obtain reductions of 12.3x, 11x and 4.65x for BFS, SSSP and PR respectively. Nonetheless, the energy savings are also significant for the TX1, achieving reductions of 5.35x, 4.54x and 1.5x for BFS, SSSP and PR respectively.

The energy savings obtained come from several sources. First, the SCU pipeline is specialized and tailored for stream compaction operations, thus being more efficient than the streaming multiprocessors' pipeline from an energy point of view. Second, the filtering operation reduces GPU workload, which further reduces dynamic energy consumption. Third, the grouping operation increase the degree of memory coalescing, which reduces the overall energy consumption of the memory hierarchy. Finally, the speedups reported in Figure 10 provide a reduction in static energy. All these factors combined allow the SCU to provide a large reduction in energy consumption.

6.2 Performance Evaluation

Figure 10 shows that the SCU provides performance improvements across all graphs and GPU systems for both BFS and SSSP graph primitives, and for PR only on the TX1. On average, we achieve speedups of 1.37x and 2.32x for the GTX980 and TX1 respectively. Performance improvement is more significant on the TX1, as it is already designed for maximum energy efficiency. We observe a speedup of 3.83x, 3.24x and 1.05x for BFS, SSSP and PR respectively. We also achieve significant speedups for the GTX980 of 1.41x, 1.65x for BFS and SSSP, although PR incurs a small slowdown. In PR all the nodes are considered active on every iteration of the algorithm, unlike BFS and SSSP. Therefore, memory accesses are less sparse and irregular, and the potential benefit the SCU is lower.

Performance benefits come from three sources. First, the SCU performs compaction operations more efficiently than the GPU, as it includes a hardware pipeline specifically designed for this task. Second, the SCU is very effective at filtering duplicated and already visited nodes, thus largely reducing the workload. Third, the grouping operation increases the degree of memory coalescing,

Table 5: Benchmark graph datasets.

Graph Name	Description	Nodes (10^3)	Edges (10^6)	Average Degree
ca [23]	California road network	710	3.48	9.8
cond [23]	Collaboration network, arxiv.org	40	0.35	17.4
delaunay [24]	Delaunay triangulation	524	3.4	12
human [23]	Human gene regulatory network	22	24.6	2214
kron [24]	Graph500, Synthetic Graph	262	21	156
msdoor [23]	Mesh of a 3D object	415	20.2	97.3

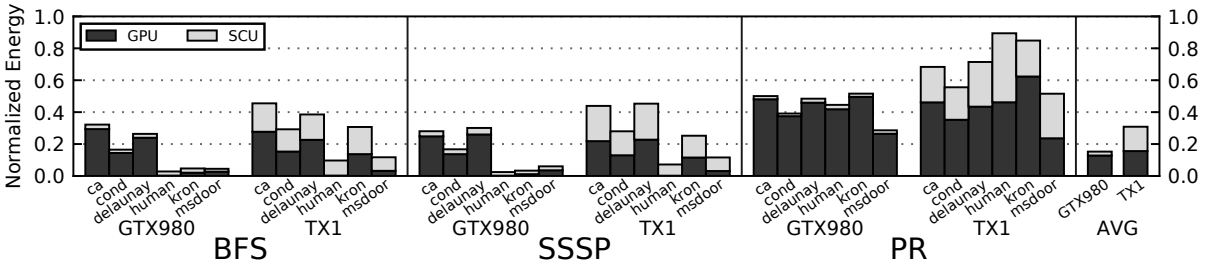


Figure 9: Normalized energy for BFS, SSSP and PR primitives on several datasets and in our two GPU systems using the proposed SCU. Baseline configuration is the corresponding GPU system (GTX980 or TX1) without the SCU. The figure also shows the split between GPU and SCU energy consumption.

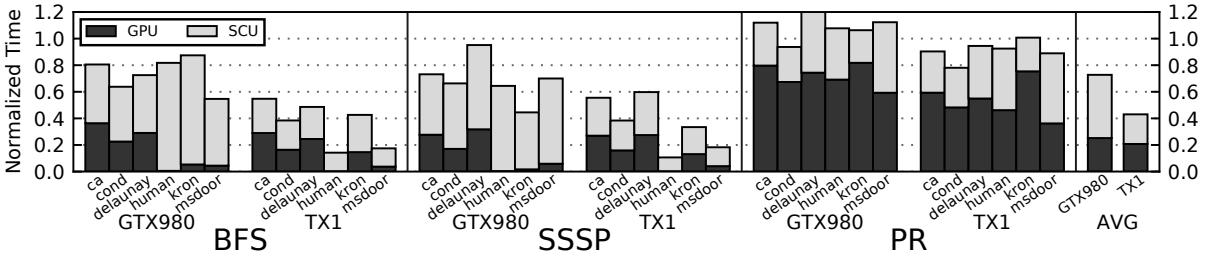


Figure 10: Normalized execution time for BFS, SSSP and PR primitives on several datasets and in our two GPU systems using the proposed SCU. Baseline configuration is the corresponding GPU system (GTX980 or TX1) without the SCU. The figure also shows the split between GPU and SCU execution time.

which reduces memory accesses and subsequently the pressure on the memory hierarchy.

6.3 Enhanced SCU Results

In Figure 11 we analyze the performance and energy benefits that result from the basic SCU presented in Section 3 and the additional benefits provided by the filtering and grouping operations described in Section 4. PR is not shown as it does not use enhanced SCU capabilities.

The basic SCU design, which is restricted to offloading compaction operations of sparse data accesses from a large graph, provides around 2x energy reduction and 1.5x speedup for both BFS and SSSP on high-performance and low-power GPUs. The enhanced SCU makes use of filtering and grouping operations that reduce GPU workload and improve memory coalescing, which leads to a reduction of memory hierarchy activity and a significant improvement over the basic SCU. The enhanced SCU achieves large energy reductions for the GTX980 of 12.3x and 11x for BFS and SSSP. For

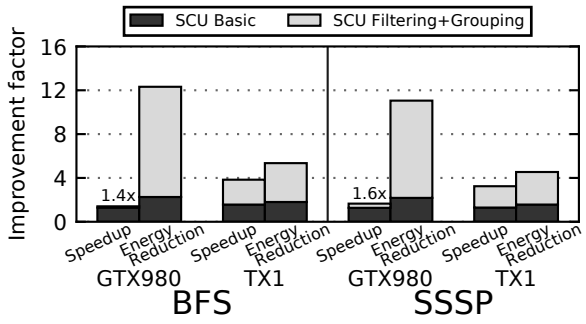


Figure 11: Speedup and energy reduction breakdown, showing the benefits due to the Basic and the Enhanced SCU.

the TX1, although being already more energy-efficient, we obtain important energy reductions of 5.35x and 4.54x for BFS and SSSP. We also achieve significant performance improvements of 3.83x and

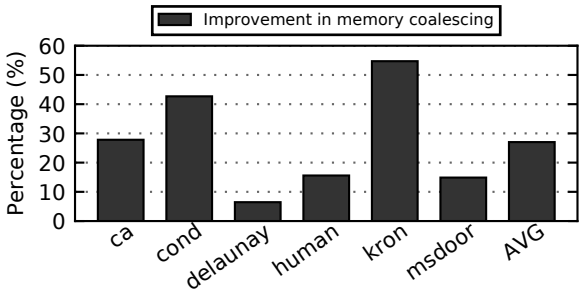


Figure 12: Improvement in memory coalescing when using the grouping operation, for SSSP primitive with TX1. Baseline configuration is SCU using only the filtering operation.

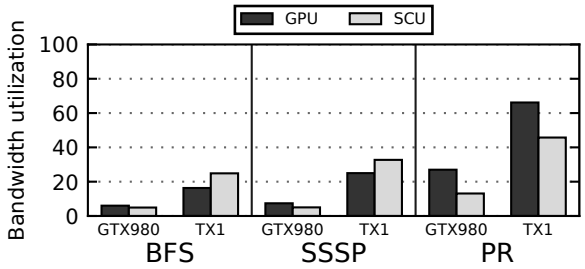


Figure 13: Memory bandwidth utilization for graph applications running on a GPU system and one incorporating the SCU. Note that each GPU system has a different bandwidth and the figure indicates utilization of peak bandwidth.

3.24x for BFS and SSSP for the TX1, whereas the improvement is lower in the GTX980 achieving a speedup of 1.4x and 1.6x for BFS and SSSP.

In BFS, the SCU is highly effective at filtering duplicated nodes, which reduces GPU workload by a large extent, improving performance and reducing GPU dynamic and static energy. For SSSP, filtering keeps only the elements with the best cost and grouping prepares the compacted data in a way that improves memory coalescing for the code executed on the streaming multiprocessors, which improves performance and reduces energy consumption on the memory hierarchy. Finally, in PR the stream compaction efforts are offloaded to the SCU without further processing, which already improves performance and provides energy reduction.

Filtering operation described in Section 4 removes duplicated and visited nodes during the compaction process with the goal to reduce the workload of the GPU by further reducing the size of the compacted array. On average, the filtering operation reduces the GPU instructions by 71% in BFS and 76% in SSSP for the TX1, with similar results for GTX980. Although the filtering operation increases the SCU workload, it reduces the workload in the GPU, resulting in important net savings in execution time and energy consumption for the overall system.

We have evaluated the efficiency of the grouping operation performed in the SCU to improve memory coalescing (see Section 4). The results for the TX1 are shown in Figure 12, where grouping improves the memory coalescing effectiveness in all the datasets, achieving an average improvement of 27%. When the required edges are being compacted, the SCU tries to write in consecutive memory locations edges whose destination nodes lie in the same cache line. By doing so, the subsequent processing of the compacted edges achieves higher GPU efficiency by reducing memory divergence. Furthermore, better memory coalescing also means that less memory requests are sent to the memory subsystem, reducing contention and overall memory traffic.

Finally, we analyze memory bandwidth usage as main memory is a constraining factor of graph applications. Graph-based algorithms expose a large amount of data parallelism, but they typically exhibit low data locality and irregular memory access patterns that prevent an effective saturation of memory bandwidth (see Section 2). Figure 13 shows how graph applications come short from saturating memory bandwidth. PR achieves higher memory bandwidth usage due to its higher regularity and data locality. Note that each GPU system has a different bandwidth and the figure indicates utilization of peak bandwidth. When comparing the GPU and SCU memory bandwidth utilization of a particular algorithm two factors come into play: performance speedup and memory accesses reduction. The high-end GTX980 system with the SCU exhibits a lower bandwidth utilization than the GPU system due to achieving higher reductions in memory accesses than in performance, whereas we see the opposite for the low power TX1 system. The TX1 system with the SCU obtains higher speedups than memory accesses reduction and, as a consequence, we see an increase in memory bandwidth utilization in BFS and SSSP.

6.4 Area Evaluation

Our results show that the SCU requires a small area of 13.27 mm^2 for the GTX980 system and 3.65 mm^2 for the TX1, including the hardware required for filtering and grouping operations. Considering the overall GPU system, the SCU represents 3.3% and 4.1% of the total area for the GTX980 and the TX1 respectively. The SCU is tightly integrated in the GPU and has access to the L2 cache which is used to store the hash table used for filtering and grouping operations, so it does not require any additional storage.

7 RELATED WORK

High performance and energy efficient graph processing has attracted the attention of the architectural community in recent years. Some previous works address the low GPU utilization problem for graph processing by modifying the GPU microarchitecture [25–27] or by optimizing the software [28–30]. On the other hand, Tigr [8] transforms the graph representation off-line to generate a more regular dataset, making it more amenable for GPGPU architectures.

Other solutions propose to replace the GPU by an accelerator devoted to graph processing. Graphicionado [31] is a customized graph accelerator based on off-chip DRAM and on-chip eDRAM, that modifies the graph data structure to optimize graph access patterns. TuNao [32] is a reconfigurable accelerator for graph processing optimized for large-scale graphs. GraphH [33] presents a processing-in-memory architecture for graph-based algorithms, whereas GraphR [34] is the first graph accelerator using ReRAM memory.

Our solution is different from previous proposals since we extend the GPU with a small programmable unit that boosts GPU performance and energy-efficiency for graph processing algorithms. We observe that, although the GPU is inefficient for some parts of graph processing (e.g. stream compaction), the phases that work on the compacted dataset can be efficiently executed on the GPU cores. Therefore, we only offload stream compaction operations to the SCU, meanwhile we leverage the streaming multiprocessors for parallel processing of the compacted data arrays. Unlike previous hardware-based solutions, our SCU has a very low cost of 3.3% and 4.1% in area for high-performance and low-power GPU respectively, and it does not require any changes in the architecture of the streaming multiprocessors.

8 CONCLUSIONS

In this paper we propose to extend the GPU with a Stream Compaction Unit (SCU) to improve performance and energy-efficiency for graph processing. The SCU is tailored to the requirements of the stream compaction operation, that is fundamental for parallel graph processing as it represents up to 55% of the execution time. The rest of the graph processing algorithm is executed on the streaming multiprocessors achieving high GPU efficiency, since it works on the SCU-prepared compacted data. We further extend the hardware of the SCU to filter out duplicated and already visited nodes during the compaction process, reducing the number of GPU instructions by more than 70% on average. In addition, we implement a grouping operation that writes together in the compacted array edges whose destination nodes are in the same cache line, improving memory coalescing by 27% for the remaining GPU workload. The end high-performance and low-power GPU designs including our SCU unit achieve speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively on average for several graph-based applications, with a 3.3% and 4.1% increase in overall area respectively.

ACKNOWLEDGEMENTS

This work was supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU).

REFERENCES

[1] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, vol. 16, no. 1, pp. 69–88, 2002.
[2] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework.," in *OSDI*, vol. 14, pp. 599–613, 2014.
[3] 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.
[4] 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.
[5] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proceedings of the GRADES'15*, p. 7, ACM, 2015.
[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, and A. Ghodsi, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, 2015.
[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. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 622–636, ACM, 2018.
[9] S. Beamer III, *Understanding and improving graph algorithm performance*. University of California, Berkeley, 2016.
[10] 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*, pp. 159–166, ACM, 2009.
[11] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, p. 14, 2015.
[12] 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*, pp. 349–359, IEEE, 2014.
[13] 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*, p. 18, ACM, 2009.
[14] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*, pp. 359–371, Elsevier, 2011.
[15] 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*, pp. 63–68, ACM, 2014.
[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.
[17] D. Compiler, "Synopsys inc," 2000.
[18] 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*, pp. 469–480, IEEE, 2009.
[19] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.
[20] 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, pp. 487–498, ACM, 2013.
[21] M. Technology, *TN-53-01. LPDDR4 Power Calculator*. Technical Report, 2016.
[22] 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*, pp. 163–174, IEEE, 2009.
[23] 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.
[24] DIMACS, "10th dimacs implementation challenge - graph partitioning and graph clustering," 2010.
[25] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, p. 3, ACM, 2011.
[26] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, vol. 46, pp. 267–276, ACM, 2011.
[27] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pp. 39–50, IEEE, 2015.
[28] 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.
[29] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 239–252, ACM, 2014.
[30] 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*, PPoPP '16, (New York, NY, USA), pp. 11:1–11:12, ACM, 2016.
[31] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–13, IEEE, 2016.
[32] 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 *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pp. 731–734, IEEE, 2017.
[33] 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*, 2018.
[34] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, Feb 2018.