2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00108

# SparseWeaver: Converting Sparse Operations as Dense Operations on GPUs for Graph Workloads

Shinnung Jeong†, Liam Paul Cooper‡, Ju Min Lee†, Heelim Choi†, Nicholas Parnenzini‡,
Chihyo Ahn‡, Yongwoo Lee†, Hanjun Kim†, and Hyesoon Kim‡
Yonsei University, South Korea†, Georgia Institute of Technology, USA‡
shin0403@yonsei.ac.kr, lpc@gatech.edu, jumin@yonsei.ac.kr, heelim@yonsei.ac.kr, nparnenzini3@gatech.edu,
ahnch@gatech.edu, dragonrain96@yonsei.ac.kr, hanjun@yonsei.ac.kr, hyesoon@cc.gatech.edu

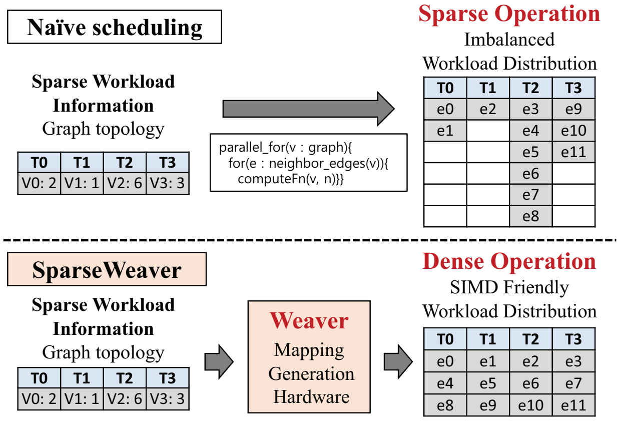
***Abstract*—Thanks to their scalable parallel processing capability, GPUs are promising computing resources for graph processing, in which identical operations are applied to a large number of edges and vertices. However, the sparsity and skewness of real-world graphs cause imbalanced workloads across GPU threads within the same warp, thus impeding efficient processing on the GPU. To mitigate this workload imbalance problem, existing works propose workload balancing hardware and software schemes. However, these solutions often suffer from additional memory overhead or increased computations and communication overheads during inter-warp and intra-warp synchronization.**

**This work proposes a new hardware-software collaborative graph processing framework, SparseWeaver, that converts sparse operations in graph processing into dense operations using graph topology and makes the workloads balanced across GPU threads. Based on the analysis of common patterns in software schemes, we propose Weaver, a new lightweight GPU functional unit microarchitecture that fully leverages the benefits of the GPU architecture and exploits memory access locality. We prototype SparseWeaver on the open-source RISC-V Vortex GPU and demonstrate 2.36 times faster execution time compared to state-of-the-art schemes while incurring a low area overhead of 0.045% from increased dedicated logic registers.**

## I. Introduction

Graphs are one of the most important and fundamental data structures for reflecting sparse relationships in the real world. Diverse analysis applications such as social network analysis, web search engines, and biological data analysis [5], [7], [9], [44], [44], [46] utilize graphs to represent their data and process graphs with various data analysis operations. Thanks to the identical operations on the large number of edges and vertices in a graph [13], GPUs have been used widely to accelerate big graph analytical processing [4], [6], [11], [12], [16], [21], [24], [30], [33], [34], [37], [38], [49].

Although GPUs offer excellent throughput through data parallelization, the sparsity and skewness of real-world graphs lead to sparse operations, which hinder efficient graph processing on GPUs. In real-world graphs, a few vertices are connected to a large number of edges [15], [18], while others are connected to only a few. Since graph processing applications gather data of neighbors for each vertex, the sparse and skewed graph causes workload imbalance across GPU threads within a warp, which makes the operation sparse. Figure 1 illustrates an example where each GPU thread, mapped to a vertex, gathers data from its neighbors via the edges connected to the vertex based on the graph topology information. Since GPUs execute all threads in a warp in lockstep, the execution time is determined by the vertex with the largest number of edges. In the naive mapping example shown in Figure 1, the warp takes six-time units to gather data from the neighbors of `v0`, `v1`, `v2`, and `v3` because `T2` requires six-time units. Although the other threads have no tasks to execute, they must wait for `T2` to gather the data before proceeding to the next task, thus wasting their computational resources.



```
parallel_for(v : graph){
  for(e : neighbor_edges(v)){
    computeFn(v, n)}}
```

Fig. 1: Workload imbalance among threads within warps during graph processing caused by irregular graph structures. SparseWeaver addresses the workload imbalance problem by converting sparse operations into dense operations, resulting in SIMD-friendly workload distributions.

To mitigate the workload imbalance problem on GPU and make sparse operations like dense operations, existing works [3], [6], [12], [23], [29], [33], [34], [37], [49], [52] propose software-based schemes. The schemes adopt data structures with indirect pointers to densely access sparse graph edges. Then, the schemes remap edges to GPU threads in a balanced way, reducing idle execution on GPU threads. However, software-only schemes still have some limitations. First, software-only schemes introduce additional overhead to remap edges across the warp or core, such as memory operation to store and share indirect pointers and additional computation to calculate mapping. Second, since the overhead of software-only schemes varies with the sparsity and skewness of a graph,

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00108

it is hard to ensure that one software-based scheme is fit for various real-world graphs [6], [21], [33]. Some existing works [32], [42], [43] propose load-balancing hardware units for generating an edge list. However, this approach suffers from additional memory access overhead while collecting edge information. Moreover, since the special hardware performs memory reading and writing for edge information itself, the hardware-based scheme cannot get the full bandwidth of the GPU architecture designs, such as hiding memory request stalls through warp-level parallelism, which can be severe for memory-intensive graph workloads.

Regarding the challenge of balancing workloads while minimizing overhead, we observe that workload imbalance can be addressed by adding a lightweight, low-overhead hardware accelerator. The fundamental reason for the workload imbalance problem is that the graph workload includes sparse operations caused by an irregular graph structure, as shown in Figure 1, but GPUs are not designed for processing sparse operations. Therefore, by adding small hardware to convert sparse operations into single instruction multiple data (SIMD)- friendly dense operations, we can effectively resolve the workload imbalance problem with minimal overhead. In addition, by integrating the hardware into the GPU execution pipeline, we can enable fine-grained and pipelined execution and leverage the benefits of the GPU architecture.

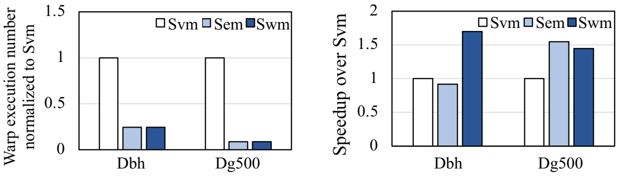
This work proposes a new hardware-software collaborative graph processing framework, SparseWeaver, that converts sparse operations in graph processing into dense operations using Weaver and balances the workloads across GPU threads. This paper makes the following key contributions:

- We propose a new hardware-software collaborative graph processing framework, called SparseWeaver, that converts sparse operations to dense operations using graph topology, making workloads balanced across GPU threads.
- Based on the analysis of common patterns in software-only existing schemes, we propose Weaver, the new lightweight hardware tightly integrated into the GPU.
- We show an in-depth evaluation of SparseWeaver compared with software and hardware schemes using real-world graph datasets and benchmarks with small hardware modifications.

## II. Background and Motivation

### A. Graph Processing on GPU

The graph processing framework on GPU [4], [6], [11], [16], [21], [24], [30], [33], [38], [49], [50] receives an input algorithm and produces analytical results of the graph. The target graph G($V$, $E$) is an abstract data structure with (vertices, $V$) and their pairs (edges, $E$). The algorithm depicts how to analyze a target graph G and generate analysis results called vertex properties. The algorithm can be expressed in three parts: how to gather and accumulate data from neighboring edges (gather and sum), how to filter active sources or destinations during edge gathering (source and destination filter), and how to update vertex properties with the gathered data (apply) [18], [21], [33], [50].



(a) Expected warp iteration

(b) Speedup over $S_{vm}$

Fig. 2: Expected warp iteration and performance of software-based scheduling schemes. (a) shows expected warp iteration using the PageRank (PR) algorithm with three different schedules ($S_{vm}$, $S_{em}$, $S_{wm}$) on the bio-human graph ($D_{bh}$) and graph-500-scale19 ($D_{g500}$). (b) shows speedup over $S_{vm}$

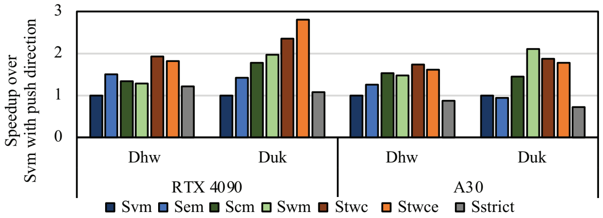


Fig. 3: Performance comparison with existing software scheduling in two Nvidia GPUs (Ampere (A30) and Ada (RTX4090)) using $D_{hw}$ and $D_{uk}$ dataset with PageRank and Auto-tuner [21]. Each graph shows speedups over $S_{vm}$.

During the gather operation, accessing neighboring edges can be performed in two steps, which leads to sparse operations. The first step is accessing the graph topology of a vertex to figure out the neighboring edges. The second step is accessing the edge information of each neighbor to perform the gather and sum operations with the information. The first step is a dense operation on vertices and can be easily parallelized across threads. However, when threads perform the second step, they must perform sparse operations for sparse neighboring edges, leading to workload imbalance and low warp utilization within warps, as shown in Figure 1.

In particular, performing gather and sum operations with real-world graphs may lead to a severe workload imbalance because of the following graph characteristics. First, real-world graphs often exhibit sparse and skewed characteristics [15], [18]. Second, a vertex can have unpredictable neighboring edges [26], [36], [41], causing irregular memory access to edge information and vertex properties. Third, some real-world graphs have millions to billions of edges, and certain vertices have a relatively large number of neighboring edges, sometimes with thousands of neighbors [1], [10], [27]. Since real-world graph characteristics exacerbate the workload imbalance problem, optimizing the performance of gather and sum operations by balancing the workload is crucial.

### B. Motivation

To address the workload imbalance problem, converting sparse operations in the second step of edge access into dense

| | **$S_{vm}$** | **$S_{em}$** | **$S_{wm}$ [33]** | **$S_{cm}$ [33]** | **$S_{twc}$ [34]** | **$S_{twce}$ [6]** | **$S_{strict}$ [12]** | **SparseWeaver** |
|---|---|---|---|---|---|---|---|---|
| Sharing Granularity | Thread | Kernel | Warp | Block | T, W, B | T, W, B | Kernel | Block |
| Imbalance | high | low | mid | low | low | mid | low | low |
| Edge Mem. Access | $2\|V\| + \|E\|$ | $2\|E\|$ | $2\|V\| + \|E\|$ | $2\|V\| + \|E\|$ | $2\|V\| + \|E\|$ | $2\|V\| + \|E\|$ | $2\|V\| + \|E\|$ | $2\|V\| + \|E\|$ |
| Shared Mem. | X | X | $3\|B\|$ | $3\|B\|$ | $3\|B\|$ | $6\|B\|$ | $3\|B\|$ | $4\|B\|$ |
| Global Mem. | X | X | X | X | $3\|V\|$ | X | $3\|V\|$ | X |
| Complexity of computing in registration | low | low | mid | mid | high | high | high | low |
| ( Sync, add Kernel, #Atom, #Warp sfhl ) | 0, 0, 0, 0 | 0, 0, 0, 0 | 1, 0, 0, 6 | 17, 0, 0, 15 | 1, 0, $3\|V\|$, 6 | 1, 3, $2\|V\|$, 0 | 17, 3, 0, 15 | 1, 0, 0, 0 |
| Complexity of computing in distribution | low | low | high | high | high | high | mid | low |
| ( #BinarySearch, #atomics, #sync ) | 0, 0, 0 | 0, 0, 0 | $\|E\|$, 0, 0 | $\|E\|$, 0, 0 | $\|E\|$, 0, 0 | 0, $\alpha\|E\|$, $\alpha\|E\|$ | $\|E\|$, 0, 0 | 0, 0, 0 |
| Edge Access Locality | low | high | mid | high | mid | mid | high | high |

TABLE I: Comparison of implementation details among existing scheduling methods [6], [12], [33], [34]. $|V|$, $|E|$, $|B|$ means the number of vertices, number of edges, and thread block size, respectively.

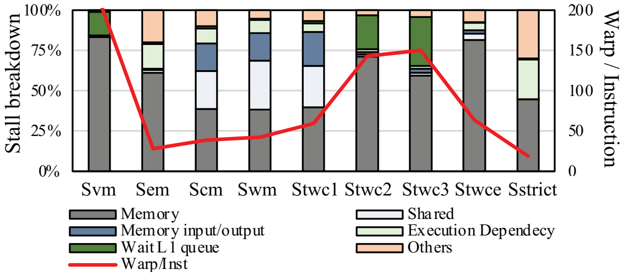


Fig. 4: Stall breakdown and warp per instruction result of existing scheduling methods with Nvidia GPU (A30) and PageRank using $D_{hw}$ dataset

operations is important, especially by remapping the edge list across threads in a dense manner. Therefore, software-based schemes [6], [12], [20], [24], [31], [33], [34], [37] suggest scheduling methods that remap active edges across threads within a warp or core to enhance the performance of edge access. Table I shows the details of various scheduling schemes. Even though the first two mapping, Vertex mapping (Naive scheduling, $S_{vm}$, mapping each vertex and its neighbor to threads and Edge mapping ($S_{em}$, mapping each edge to thread), seem simple, they can suffer from workload imbalance and twice the memory access for edge information. More complex scheduling schemes ($S_{wm}$ [33], $S_{cm}$ [33], $S_{twc}$ [34], $S_{twce}$ [33], and $S_{strict}$ [12]) try to balance workloads by sharing graph topology across blocks, warps, or threads. These scheduling methods use additional computation, synchronization, and shared/global memory access to achieve better performance by concealing overhead with the benefits gained from resolving the workload imbalance. We will elaborate in Section III-A.

However, balancing the workload while minimizing overhead still remains a challenge because the overhead is not always successfully hidden. Figure 2 shows the expected number of warp iterations and performance for the edge-gathering process when executing PageRank application with $D_{bh}$ and $D_{g500}$ graph dataset [40]. As shown in Figure 2a, due to workload imbalance, $S_{vm}$ requires 4x and 11x more warp iterations compared to $S_{wm}$ and $S_{em}$ for the $D_{bh}$ and $D_{g500}$. Even though $S_{wm}$ and $S_{em}$ show similar expected warp iteration by balancing the workload, the performance can vary because of different overheads. $S_{em}$ needs double the memory reads to get edge data, while $S_{wm}$ uses additional computation and shared memory. Figure 2b, shows the best performance with $S_{wm}$ and $S_{em}$ for the $D_{bh}$ and $D_{g500}$ datasets. Because $D_{g500}$ has relatively more vertices and fewer edges than the $D_{bh}$ graph, it incurs relatively more overhead from additional computations per edge.

The workload imbalance problem also occurs in Nvidia GPUs. Figure 3 shows that complex software scheduling methods ($S_{cm}$, $S_{wm}$, $S_{twc}$, $S_{twce}$) often perform better than $S_{vm}$ in Nvidia GPUs. By changing the scheduling, a maximum of 2.80x speedup can be achieved. However, these scheduling methods also introduce overhead on Nvidia GPUs, as shown in Figure 4 [1] . Note that PageRank consists of one addition and a read and write for the edge and vertex. Nevertheless, some scheduling methods introduce other stalls such as shared memory stalls ($S_{wm}$, $S_{cm}$, and $S_{twc1}$), L1 queue waiting stalls ($S_{vm}$ or $S_{twc1}$), or higher warp/instructions. These graphs show that determining the best scheduling scheme for each dataset and application is hard , and scheduling methods often incur some overhead. Some existing works [6], [21], [33] have suggested an auto-tuner for graph processing on GPUs, but this approach is time-consuming and incurs abstraction costs.

Thus, the question arises: *Can software optimization be considered a fundamental solution to this problem?* We observe that the fundamental problem of processing graph workloads on GPU is the difficulty in directly mapping irregular data-dependent graph tasks onto the one-dimensional parallel units of GPU. Existing software-based works can be considered to distribute irregular data-dependent graphs to GPU warps at runtime (scheduling) or static time (storage format) to achieve workload balance. However, the software-based works require additional computations and synchronizations because each thread must generate and compute the information needed by itself. Exploring efficient scheduling or storage formats using DSE (Design Space Exploration) tools [21] helps identify the best scheduling option among many. Nevertheless, the workload imbalance problem remains because GPU hardware

[1] The original stall metric names in Nvidia Nisight Compute are the following: Memory (long scoreboard), Shared (short scoreboard), Memory input/output (MIO Throttle), Execution Dependency (Wait), Wait for L1 queue (LG Throttle), and Warp/Instruction (average warp latency per instruction issued).

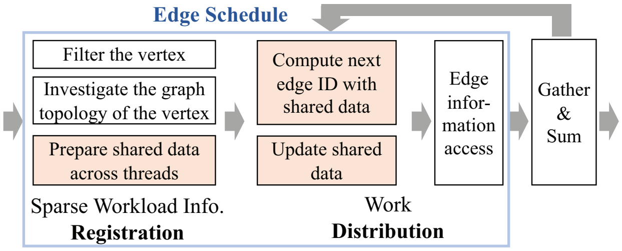


Fig. 5: Disassembly of graph processing through abstraction of the software-based scheduling scheme

does not support such data-dependent workload mapping in its one-dimensional structure.

To fundamentally solve the workload imbalance problem, the hardware needs to convert sparse operations into dense operations. Since hardware can generate dense edge mapping with a smaller overhead compared to the overhead of software schemes caused by redundant calculations and synchronized updates while each thread calculates the mapping itself. In addition, the conversion can be accelerated by implementing the remapping process through a separate FSM. However, designing conversion hardware has several challenges. First, simply offloading the entire process of neighboring edge access can miss possible benefits from the fine-grained pipeline and microarchitecture of GPU. Second, the conversion hardware can be a bottleneck when performing edge data access with memory-intensive graph workloads. Third, the offloading can cause additional memory usage and access.

Therefore, in this paper, we analyze the existing software-based schemes and identify the key part of the conversion of sparse operations into dense operations. Then, we suggest a new lightweight hardware extension that is tightly integrated into the GPU execution pipeline, leveraging the benefits of the GPU architecture.

## III. Hardware/Software Co-Design

### A. Software-based Scheduling Abstraction

Existing software-based scheduling schemes [6], [12], [33], [34] have common patterns during mapping generation. These schemes collect graph topology and generate edge identifiers across warps, blocks, or entire kernels. As shown in Figure 5, the software-based scheduling scheme can be divided into two stages: (1) the **registration stage** and (2) the **distribution stage**. In the registration stage, each thread registers some data, such as sparse workload information, to use in the distribution stage. Each thread prepares the information by filtering the base vertex ID (source vertex or destination vertex ID), accessing the graph topology of the base vertex, and computing additional information. In the distribution stage, each thread performs the work based on sparse workload information. Each thread generates an edge ID used in the current lockstep based on shared data. After finishing lockstep, the thread updates shared data for the next iteration for itself and other threads. The thread performs edge information access, gather, and sum based on the generated indirect pointers.

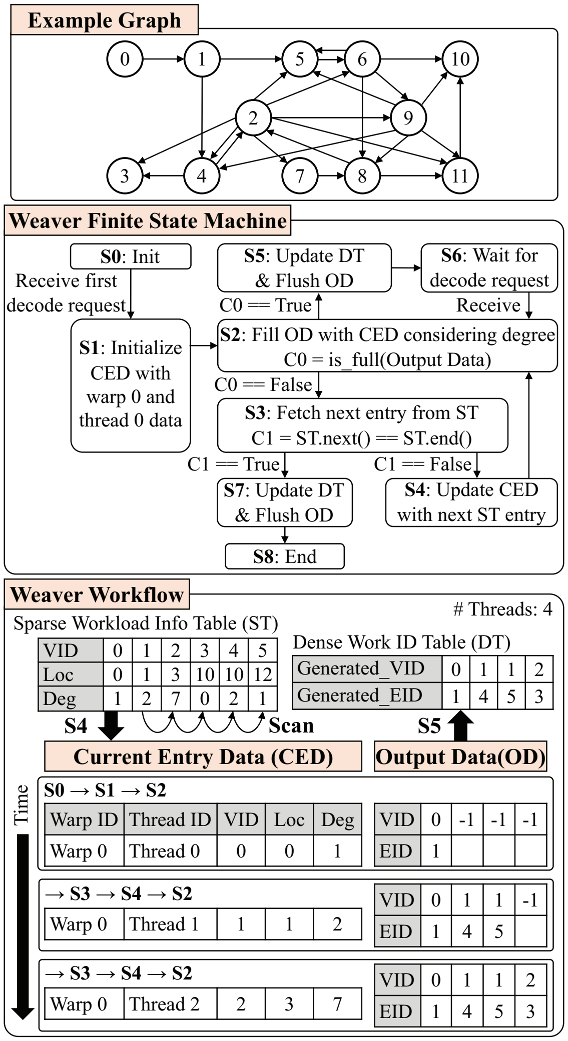
Table I shows details of the existing scheduling scheme. Based on the target scheduling granularity, each scheduling has different additional overheads, such as memory usage and computations of the registration and distribution stage. For example, the warp-sharing mapping ($S_{wm}$) balances the workload within each warp. In the registration stage, each thread loads the graph topology to figure out the degree of the neighbor edge and the edge indicator of the first neighboring edge. Then, each thread within the warp generates a prefix sum array of degrees and updates the array in the shared memory. In the distribution stage, each thread calculates an edge indicator for itself by performing a binary search on the degree prefix sum. This process results in $O(nlog(n))$ time complexity for shared memory scans.

Software schemes employ various levels of balance granularity because generating finer-grained mapping not only brings more balance but also increases overhead. Balancing across kernels [12] is expensive due to more costly global memory access overhead or additional kernel launch overhead. Therefore, other schemes [6], [33], [34] try to achieve better performance using warp-level and block-level balancing, recognizing that block-level sharing introduces additional overhead for synchronization and searching. Unlike software-based schemes, adding hardware to each core eliminates concerns about the overhead of block-level sharing. Since the GPU inherently executes one warp at a time and needs to dynamically return the edge indicator for the executing warp at that moment, we aim to design hardware that achieves block-level workload balancing.

As shown in Figure 5, the core part of existing scheduling methods is using sparse workload information to evenly distribute edge ID in a dense manner for threads within warps. Offloading the minimal yet core parts of the scheduling to lightweight hardware might help reduce overhead and accelerate the process. With well-designed hardware, GPU can help reduce additional buffers for scheduling and conceal scheduling process overhead in the GPU pipeline. Therefore, we only decide to offload the following part emphasized as orange boxes in Figure 5: (1) prepare shared data across the threads, (2) compute the next edge indicator with shared data, and (3) update shared data.

### B. Weaver Logic Design

SparseWeaver generates an edge indicator and maintains shared data. Therefore, SparseWeaver employs Sparse Workload Information Table (ST) and Dense Work ID Table (DT) to keep shared data from registration and output data for distribution, respectively. The ST stores shared registration data such as vertex ID (VID), degree of neighbors, and the start location of the neighbor edges in the neighbor edge array. In the registration stage, SparseWeaver collects shared data and fills the ST. The DT stores Work IDs such as base vertex ID and generated edge ID (EID) (location in neighbor edge


Example Graph
0
1
2
3
4
5
6
7
8
9
10
11
Weaver Finite State Machine
S0: Init
Receive first decode request
S1: Initialize CED with warp 0 and thread 0 data
S2: Fill OD with CED considering degree
C0 = is_full(Output Data)
C0 == True
S5: Update DT & Flush OD
S6: Wait for decode request
Receive
C0 == False
S3: Fetch next entry from ST
C1 = ST.next() == ST.end()
C1 == True
C1 == False
S7: Update DT & Flush OD
S4: Update CED with next ST entry
S8: End
Weaver Workflow
# Threads: 4
Sparse Workload Info Table (ST)
Dense Work ID Table (DT)
S4
Scan
S5
Current Entry Data (CED)
Output Data(OD)
Time
S0 → S1 → S2
→ S3 → S4 → S2
→ S3 → S4 → S2

| VID | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Loc | 0 | 1 | 3 | 10 | 10 | 12 |
| Deg | 1 | 2 | 7 | 0 | 2 | 1 |

| Generated_VID | 0 | 1 | 1 | 2 |
|---|---|---|---|---|
| Generated_EID | 1 | 4 | 5 | 3 |

S0 → S1 → S2

| Warp ID | Thread ID | VID | Loc | Deg |
|---|---|---|---|---|
| Warp 0 | Thread 0 | 0 | 0 | 1 |

| VID | 0 | -1 | -1 | -1 |
|---|---|---|---|---|
| EID | 1 | | | |

→ S3 → S4 → S2

| | | | | |
|---|---|---|---|---|
| Warp 0 | Thread 1 | 1 | 1 | 2 |

| VID | 0 | 1 | 1 | -1 |
|---|---|---|---|---|
| EID | 1 | 4 | 5 | |

→ S3 → S4 → S2

| | | | | |
|---|---|---|---|---|
| Warp 0 | Thread 2 | 2 | 3 | 7 |

| VID | 0 | 1 | 1 | 2 |
|---|---|---|---|---|
| EID | 1 | 4 | 5 | 3 |

Fig. 6: SparseWeaver hardware logic called Weaver

array). In the distribution stage, SparseWeaver returns work IDs in the DT for edge information access, gather, and sum.

In SparseWeaver, Weaver generates Work IDs for the warp by scanning the registered ST and decoding its entries. Weaver loads and maintains the current-looking ST entries in a buffer called Current Entry Data (CED). Then, Weaver generates the intermediate Work IDs by decoding CED and maintains the intermediate data in a buffer called Output Data (OD). Figure 6 shows the **Finite State Machine (FSM)** and workflow of Weaver. The logic can fill an OD buffer from multiple low-degree entries (S3 → S4 → S2) and also can fill multiple OD buffers from a high-degree entry (S5 → S6 → S2). When Weaver receives the first decode request for edge ID, Weaver is initialized with the first entry of the CED buffer (S1), and Weaver tries to fill out every entry of OD simultaneously with the CED information (S2). If the degree of the CED is not enough to fill the OD (C0 == False), Weaver fetches and updates the next entry of ST (S3, S4), then tries to fill OD again (S2). If all entries in OD are filled, FSM updates DT (S5) and waits for the next decode request (S6). This mechanism can handle high-degree entries by filling multiple OD buffers. If all ST entries are scanned, the FSM goes to the end states (S7, S8). When Weaver status is end, SparseWeaver returns empty Work IDs (such as -1). The Weaver FSM is initialized to init status when a new registration request is received in the registration stage.

For example, Figure 6 shows how SparseWeaver generates Work IDs, assuming that the number of threads within a warp is 4. First, Weaver reads the first entry in the ST and fills the CED buffer, which has vertex ID 0, start location 2, and degree 1 in initialization stages (S0 → S1). Then, Weaver fills one entry of the OD buffer with vertex ID 0 and edge ID 2 because the degree of the CED entry is 1 and the start location is 2, in a decoding stage (S2). Because a low degree entry is not enough to fill the OD buffer, Weaver fetches the next entry of ST and updates the CED buffer (S3, S4). Since the OD buffer has three remaining entries, Weaver loads new ST entries (2, 10, 2) and fills two entries of the OD buffer as (2, 10) and (2, 11). Again, Weaver loads the next ST entry (4, 30, 5), fills the final OD buffer entry with Work ID (4, 30), and finally goes to S5 in the state machine.

### C. SparseWeaver Design

To tightly integrate Weaver into the GPU pipeline, we design the execution workflow of SparseWeaver with Weaver. In this subsection, we discuss design decisions to integrate Weaver into the GPU execution flow.

**SparseWeaver Input** To achieve our final goal of converting sparse operations into dense operations, Weaver must receive shared data such as vertex ID, location, and degree. Since the storage format inherently has information about incoming and outgoing edges for a given vertex, SparseWeaver can gather incoming or outgoing information for a given workload in the registration step. Whether investigating incoming edges (pull direction) or outgoing edges (push direction), shared data—such as the base vertex ID, start location of the neighbor edge list, and degree—can be collected [21]. SparseWeaver registers shared data into a ST for distribution stage. Since the GPU cores perform the graph topology access and the data is stored in the registers, SparseWeaver requires an ISA extension to delineate the extraction of gathering data from the destination registers of certain code sequences.

†**Design Decision**: Gather workload information from the GPU core: Base vertex ID, Location, and Degree.

**SparseWeaver Output** SparseWeaver generates work IDs for each thread within a warp simultaneously for every decode request. To deliver these Work IDs to the GPU core, SparseWeaver should return Edge IDs by a new ISA. In addition, SparseWeaver returns the base vertex ID to indicate one vertex in the edge vertex pair, facilitating fast-edge data access [21]. This process also needs to deliver work to the core to enable access to the following instructions, requiring another ISA extension to take work IDs from SparseWeaver to the GPU core.

Additionally, SparseWeaver offers a thread mask to provide clues about thread activation. Basically, SparseWeaver activates all the threads within the core and makes them perform the distribution stage. During mapping, SparseWeaver can generate a map where some threads may not have assigned work. For example, there are cases where the total degree is not evenly divided by the number of threads in a warp. Although the graph needs to either perform or skip work, thread divergence can occur and can cause performance issues, such as leading to the need for divergence control logic like split and join [48]. Therefore, by returning the thread mask, SparseWeaver provides a clue for replacing this logic with hardware-controlled active threads.

**†Design Decision**: Return workload indicator to GPU core: EID and VID Return a clue for thread activation.

**Out of Order Registration and Ordered Scan** SparseWeaver requires the ST not only to buffer shared data but also to facilitate entry access ordered by VID. During the gathering process, the vertex ID ordered edge access can enhance performance [33]. However, due to the out-of-order execution of warps, SparseWeaver may receive data in an unordered manner. Therefore, we use a two-step method to generate an ordered ST. First, the kernel code generated by the SparseWeaver compiler includes investigation code that uses software thread IDs (e.g., CUDA thread ID or global ID of OpenCL), enabling each thread to investigate a vertex in order. Second, we use the warp ID and thread ID as keys to store shared data in the ST.

**†Design Decision**: Perform ordered decode. For ordered decode, use software thread ID to investigate the graph topology in the kernel code and use hardware thread ID and warp ID to index the workload information table.

**Dynamic Work Distribution** SparseWeaver dynamically distributes the work IDs, supporting an out-of-order warp execution-friendly distribution. More precisely, SparseWeaver maps and distributes the work according to the order in which requests are received rather than by warp ID. Dynamic distribution can improve locality by maintaining similar edge IDs currently in flight, which can aid in optimal warp execution.

**†Design Decision**: Distribute the workload dynamically to maximize the benefit from graph locality.

**Synchronization between Registration and Distribution** SparseWeaver requires synchronization for three reasons. First, SparseWeaver needs to ensure that all warps have registered the data they have to process. The synchronization is necessary because only then can the ST be completed based on the workload information data by all warps in the core, and an ordered scan can be guaranteed. Second, the occurrence of registration is decided by graph topology, meaning the number of vertices that warps need to process in each iteration varies depending on the data. Therefore, Weaver cannot predict how many warp requests will occur, requiring explicit synchronization points in GPU code. Third, this synchronization is important for preventing early exits. For example, if a specific warp exits the registration and distribution stages before another warp has completed registration, it cannot maximize the utilization of warp resources to perform the distribution stage with the remaining workload. For these reasons, we add synchronization between the registration step and the distribution step. This process is also necessary for other complex scheduling algorithms, and we require just one synchronization per examination. Note that software-based schemes also have at least one synchronization except for native schemes.

**†Design Decision**: Insert synchronization between the registration stage and the distribution stage in the GPU code.

**Filtering workload** Filtering is crucial for performance in some graph applications [50], so the work IDs distributed by SparseWeaver must be filtered wherever possible. This means that we should return a filtered work ID when filtering for base vertex ID is feasible. However, the main focus of SparseWeaver is on distributing work IDs, and it aims to minimize additional structures and logic by not addressing issues that can be resolved within the GPU pipeline. Consequently, the SparseWeaver compiler should insert filters to the registration stage when they are associated with the base vertex ID. Then, SparseWeaver inserts code that changes the degree to zero when a vertex is filtered, so no related work IDs are generated during the distribution stage.

Furthermore, there are algorithms like BFS that do not need to process remaining neighbors during gather processing once the needed information has been collected. In such cases, there may be a need for an early exit from the distribution stage for a specific vertex ID. This is particularly important for nodes like supernodes that have a high number of neighbors, where decoding might need to be stopped midway. Therefore, SparseWeaver requires an ISA to signal to Weaver that specific vertex ID doesn't be decoded anymore and can be skipped.

**†Design Decision**: Integrate a filter into the GPU code at the appropriate location and send a skip signal when no further distribution is needed for a specific vertex.

### D. Assembling Design Decisions

Figure 7 illustrates the overall workflow of SparseWeaver based on our design decisions, demonstrating how warps are executed over time and how tasks can be offloaded to SparseWeaver. In the GPU core, execution proceeds in the order of registration, synchronization, and distribution.

In the registration stage, the GPU core investigates shared data such as vertex ID, location, and degree to deliver that information to Weaver. Since we assume out-of-order warp execution, Weaver stores shared data into the Sparse Work Information Table indexed by warp ID and thread ID. For example, three warps in Figure 7 are executed in the order of

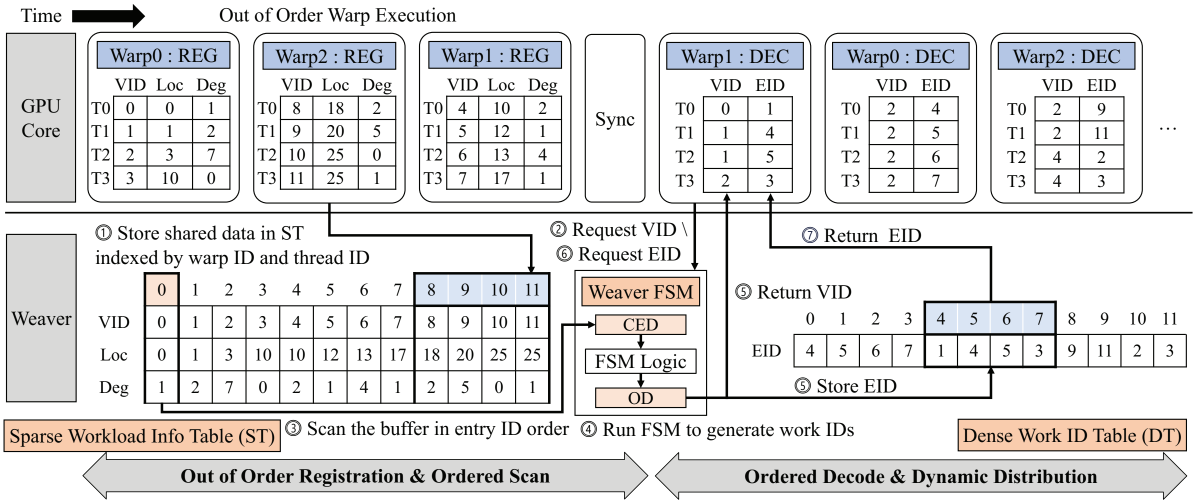

Time
Out of Order Warp Execution
GPU Core
Warp0 : REG
Warp2 : REG
Warp1 : REG
Sync
Warp1 : DEC
Warp0 : DEC
Warp2 : DEC
VID Loc Deg
VID EID
T0
T1
T2
T3
…
Weaver
① Store shared data in ST indexed by warp ID and thread ID
② Request VID \
⑥ Request EID
⑦ Return EID
⑤ Return VID
⑤ Store EID
Weaver FSM
CED
FSM Logic
OD
VID
Loc
Deg
EID
③ Scan the buffer in entry ID order
④ Run FSM to generate work IDs
Sparse Workload Info Table (ST)
Dense Work ID Table (DT)
Out of Order Registration & Ordered Scan
Ordered Decode & Dynamic Distribution

Fig. 7: SparseWeaver execution flow. SparseWeaver receives workload information (VID, offset, deg) from the GPU warp and stores it in a Sparse Workload Information Table indexed by warp ID and thread ID. If SparseWeaver receives a request to distribute VID from a warp, SparseWeaver scans the buffer in entry ID order and generates vid and eid for that warp. SparseWeaver first returns VID and stores EID in the Dense Work ID Table. Then, SparseWeaver returns EID loaded from the Dense Work ID table when receiving a request from warp by indexing warp ID.

| Instruction | IType | Opcode | funct | Description |
|---|---|---|---|---|
| WEAVER_REG, VID, loc, deg | C | CUSTOM1 | 1 | Register VID, loc, deg |
| WEAVER_DEC_ID, VID | R | CUSTOM0 | 7 | Return VID of next workload |
| WEAVER_DEC_LOC, EID | R | CUSTOM0 | 8 | Return EID of next workload |
| WEAVER_SKIP, VID | C | CUSTOM1 | 2 | Send skip signal using VID |

TABLE II: SparseWeaver instructions

Warp 0, 2, 1, and the shared data of Warp 2 is stored in the entries 8, 9, 10, and 11 of the ST. If a vertex is filtered, the thread that investigates that vertex changes the degree to 0. After finishing the registration stage, every active warp waits for other warps in the synchronization point.

In the distribution stage, the GPU core sends decode requests to get work IDs, and Weaver generates work IDs. As shown in Figure 7, Warp 1 sends the first decode request for VID to Weaver. Identical to FSM in Figure 6, Weaver reads ST entries (0, 2, 1), (2, 10, 2), and (4, 30, 5) and generates OD (0, 2), (2, 10), (2, 11), and (4, 30). Then, Weaver returns VID (0, 2, 2, 4) to the GPU core and stores EID (2, 10, 11, 30) in the Dense Work ID table. If Warp 1 makes a decode request for edge ID, Weaver returns edge ID (2, 10, 11, 30). Warps can filter opposite vertex ID by accessing that using edge ID. By assembling design decisions in Section III-C, SparseWeaver can integrate Weaver into GPU execution flow successfully.

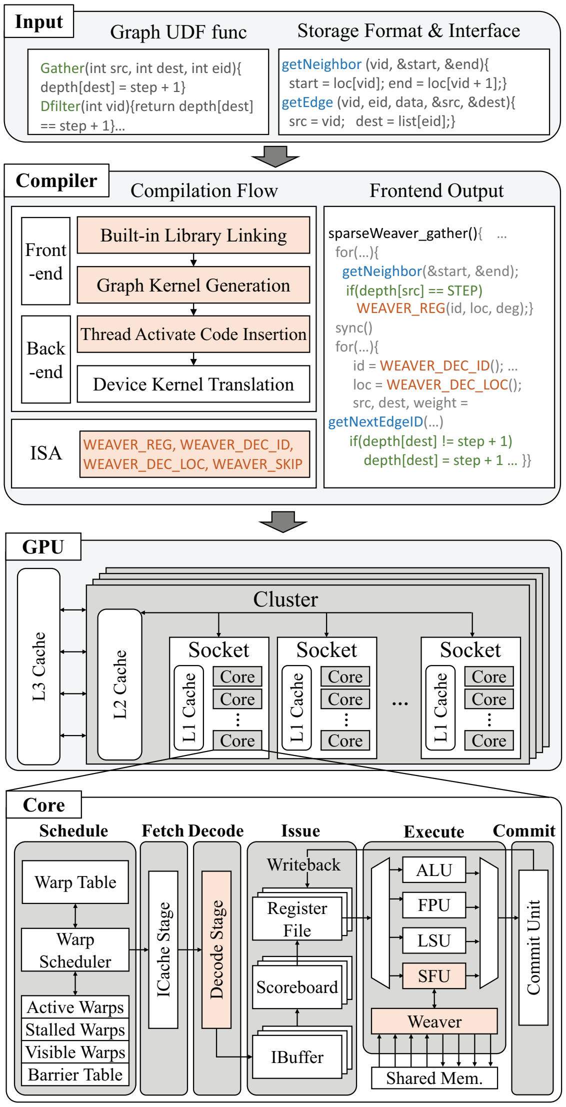
Notably, SparseWeaver supports storage formats where edges are stored consecutively, and sparse workloads are indicated in the offset array by neighbor counts such as CSR, Tigr, or CR2. These formats rely on base vertex IDs to access the offset array and edge list. In addition, SparseWeaver can accommodate non-consecutive labeling by splitting vertices and registering split vertices as separate entries. Since SparseWeaver receives vertex ID, which can support any order of vertices. SparseWeaver can also handle hybrid formats like ELL by applying its functionality to the CSR subgraph, making it versatile for scheduling even in non-consecutive or tiled storage scenarios.

## IV. SparseWeaver Framework

Figure 8 shows a system overview of SparseWeaver. The SparseWeaver system receives input User Defined Functions (UDFs) for algorithm and graph using a storage format, storage format interface, and direction. The UDFs consist of four different methods: *init*, *gather*, *apply*, and *filter*. A user breaks down a graph algorithm to implement each method like other graph processing frameworks [21]. Since the input graph is stored using a specific storage format, such as Compressed Sparse Row format (CSR), the user uses the storage format interface to let SparseWeaver access the storage format. The storage format interface has two methods [21], *getNeighbor* and *getEdge*, to access graph topology and edge information. The direction represents whether the gathering process collects incoming or outgoing edges. The SparseWeaver frontend compiler receives the graph algorithm and generates GPU kernel code. The SparseWeaver backend compiler performs target-specific code optimization and generates GPU binary. The SparseWeaver GPU executes the compiled binary with Weaver support. We implement Weaver on the Open-source RISC-V GPU called Vortex [14], [47], [48], and use PoCL [22] and LLVM [28] as the frontend and the backend compiler.

### A. SparseWeaver Instruction

Table II shows the SparseWeaver instructions, which serve as the interface for passing inputs and outputs to and from the Weaver. The SparseWeaver has a WEAVER_REG instruction to register the base vertex ID, start edge ID, and



```
Gather(int src, int dest, int eid){
depth[dest] = step + 1}
Dfilter(int vid){return depth[dest]
== step + 1}...
```

```
getNeighbor (vid, &start, &end){
  start = loc[vid]; end = loc[vid + 1];}
getEdge (vid, eid, data, &src, &dest){
  src = vid;   dest = list[eid];}
```

```
sparseWeaver_gather(){   ...
  for(...){
   getNeighbor(&start, &end);
    if(depth[src] == STEP)
      WEAVER_REG(id, loc, deg);}
  sync()
  for(...){
     id = WEAVER_DEC_ID(); ...
     loc = WEAVER_DEC_LOC();
     src, dest, weight =
getNextEdgeID(...)
    if(depth[dest] != step + 1)
      depth[dest] = step + 1 ... }}
```

Fig. 8: SparseWeaver system overview

degree. Additionally, SparseWeaver has two output instructions to assist in fetching data from SparseWeaver microarchitecture: WEAVER_DEC_ID for returning the vertex ID and WEAVER_DEC_LOC for edge ID. The SparseWeaver also has WEAVER_SKIP instruction to skip the vertex if it is still decoding. Since we implemented our work on the RISC-V-based open-source GPU, instructions are implemented in RISC-V format. Based on the RISC-V manual [51], WEAVER_DEC_ID and WEAVER_DEC_LOC are implemented as R-type instruction, formed opcode, rd, funct3, rs1, rs2, func7 and WEAVER_REG is implemented as CUSTOM instruction in the form of opcode, rd, funct3, rs1, rs2, funct2, r3. We use funct3 and funct2 to distinguish instruction.

The important point is that one of the output instructions needs to indicate the exit of the work distribution loop. Although the thread mask can control whether tasks proceed, the thread mask cannot represent the exit status because the deactivation of all threads can cause the termination state. Thus, the WEAVER_DEC_ID instruction returns -1 to indicate the exit of the loop. If all activated threads return -1, then SparseWeaver can reach the exit the distribution stage.

```
 1 void SparseWeaverGatherKernel(wset, graph) {
 2     tid = get_thread_id() // Get the TID
 3     // Registration step
 4     for (id = wset.base_id + tid; id < wset.bound;
       id += wset.stride) {
 5         vid = getFrontier(id)
 6         if (dest_filter(vid))
 7             continue
 8         start, end = getNeighbor(graph, ...)
 9         WEAVER_REG(vid, start, end - start)}
10     synchronization()
11     while (true) { // Distribution step
12         vid = WEAVER_DEC_ID()
13         if (vid == -1)
14             break
15         eid = WEAVER_DEC_LOC()
16         src, dest, weight = getEdge(eid...)
17         if (dest_filter(dest))
18             WEAVER_SKIP(dest)
19         if (src_filter(src))
20             continue
21         computeFn(loc, src, dest, weight)
22     }
23 }
```

Fig. 9: Generated SparseWeaver gather kernel (Pull)

### B. *SparseWeaver Compiler*

With the updated PoCL and LLVM enabling the adoption of the Weaver ISA and device-specific runtime libraries, the compilation flow proceeds through distinct frontend and backend stages, each responsible for critical optimizations that tailor the kernel to the target architecture.

The SparseWeaver frontend compiler receives inputs and generates graph processing kernels like Figure 9. The frontend compiler performs two optimizations. First, the frontend compiler performs **Built-in Library Linking**, such as atomic or math functions, which are linked according to the hardware target. Secondly, after linking, the compiler performs **Graph Kernel Generation** by combining Weaver ISA intrinsic, user-defined functions and storage format interface. Figure 9 illustrates an example of a pull-direction kernel code generated by the frontend compiler, which traverses incoming edges based on the destination vertex ID to perform gather processing. The kernel executes the registration stage first (lines 4–9) and then the distribution stage (lines 11–22). Kernel code includes the WEAVER_REG intrinsic to deliver shared data to Weaver in the registration stage (line 9) and the WEAVER_DEC_ID and WEAVER_DEC_LOC intrinsic to decode work IDs in the distribution stage (line 12 and line 15). Depending on the direction, the compiler places filters in the registration and distribution stages. In the case of a pull direction, the compiler inserts an additional destination filter at the registration stage to insert the WEAVER_SKIP intrinsic (line 18).

| Graph Name | Number of Nodes | Number of Edges |
|---|---|---|
| bio-human-gene1 ($D_{bh}$) | 22,284 | 24,691,926 |
| bio-mouse-gene ($D_{bm}$) | 45,102 | 29,012,392 |
| roadNet-CA ($D_{rn}$) | 1,971,282 | 553,321 |
| road-central ($D_{rc}$) | 14,081,817 | 3,386,682 |
| graph500-scale19 ($D_{g500}$) | 335,319 | 15,459,350 |
| COLLAB ($D_{co}$) | 372,475 | 49,144,316 |
| hollywood-2011 ($D_{hw}$) | 2,180,653 | 228,985,632 |
| web-uk-2005 ($D_{uk}$) | 129,633 | 23,488,098 |
| web-wikipedia ($D_{wk}$) | 2,936,414 | 104,673,033 |

TABLE III: Graph dataset information [40]

The SparseWeaver backend compiler receives the kernel IR generated by the frontend compiler and performs **Thread Activate Code Insertion** before translating it into a GPU binary through device kernel translation. Target-specific code optimization inserts thread mask control logic to activate all threads within warp during performing distribution and remaining gather steps. For example, on the Vortex GPU, the backend compiler inserts code that stores the thread mask of warp and activates all threads before entering the distribution loop (line 11 in Figure 9). Then, the backend compiler inserts code that restores the thread mask right after finishing the distribution loop. Additionally, the backend compiler **expands the ISA Table** to include Table II, supporting kernel translation. With compiler support, SparseWeaver generates the GPU code necessary to execute graph code.

### C. Weaver Implementation

SparseWeaver is implemented on top of the Vortex GPU by extending the Special Function Unit (SFU) with Weaver as shown in Figure 8. When the GPU core decodes and issues Weaver instructions, the GPU core can send a register request or decode request to Weaver. The Sparse Work Information Table and Dense Work ID Table of Weaver can be implemented using either registers or shared memory. We implement tables using shared memory based on the following reasons. The Vortex GPU has a limited number of registers and fast shared memory access. In addition, SparseWeaver has a relatively small number of table accesses (only for one read and write per graph topology data) while processing a relatively large number of edges, so table accesses might not cause high overhead and can be concealed successfully by GPU pipeline execution. We will discuss this in Section V-D.

## V. Evaluation

In this section, we evaluate the performance of SparseWeaver with four graph algorithms operators and GCN on nine graphs, comparing with the four software scheduling schemes ($S_{vm}$, $S_{em}$, $S_{wm}$, $S_{cm}$ [33]) and edge-generating hardware (a similar approach to hardware-based scheme [32], [42], [43]) on an open-source RISC-V GPU called Vortex [14], [47], [48]. We model SparseWeaver on top of the Vortex GPU that provides an open-source software stack, cycle-level simulator, and register transfer level (RTL) hardware description. Thanks to its own simulator, Simx, achieving cycle accuracy within 6% compared to the RTL model [14], we can test the performance of SparseWeaver. In addition, we also extended LLVM [28] and PoCL [22] for Vortex [19] to support Weaver ISA and implement compiler optimizations. In this evaluation, we use Vortex hardware configurations such as 2 sockets, 3 cores per socket, 32 warps per core, and 32 threads per warp. The L1 and L2 cache sizes are set as 64KB and 1MB, respectively. A penalty is applied for testing with SparseWeaver, reducing the L1 cache size to 32KB for using 512 entries in the Sparse Work Information table and the Dense Work ID table per core. To examine the hardware overhead, we extended Vortex RTL and synthesized using Quartus Prime Pro 8.1, targeting Intel Stratix 10 FPGA. The evaluation uses nine graphs in Table III. This evaluation uses four different algorithms such as PageRank (PR) [8], Connected Components (CC) [45], Breadth-First-Search (BFS) [35] and Single Source Shortest Path (SSSP) [35].

### A. Comparison with Software-based Schemes

Figure 10 shows that SparseWeaver improves performance for most algorithms. SparseWeaver achieves 2.36x speedups for the vertex mapping and 2.63x speedups for the edge mapping. SparseWeaver achieves 2.73x, 2.64x, 2.71x, and 1.60x speedups over the $S_{vm}$ for BFS, SSSP, PR, and CC.

In detail, SparseWeaver outperforms all other software scheduling schemes across four benchmarks. It is particularly notable in the speedup achieved through the BFS and SSSP benchmarks. These benchmarks have destination and source filters, which result in higher load imbalances among vertices, thereby allowing SparseWeaver with BFS and SSSP to demonstrate superior performance compared with other scheduling schemes. BFS shows more speedup than SSSP because it does not use edge weight information. PR and CC perform for all edges in the gather step, resulting in better opportunities to benefit from workload balance. In addition, the balanced workload enables coalesced memory access during edge information access for the PR algorithm, whereas $S_{vm}$ does not achieve efficiency from this approach. Therefore, with PR, all other scheduling schemes show performance improvement over $S_{vm}$. The benchmark CC algorithm employs an apply kernel to rapidly propagate connection IDs among connected components. The component IDs are composed of nearby IDs due to the reordering of benchmark graphs to reveal community structures. If no updates are required, the CC algorithm simply checks the IDs of the source and destination, resulting in minimal imbalance costs. Consequently, SparseWeaver exhibits relatively small performance enhancement due to the concealed benefits of scheduling in the gather kernel and the proportion of the total time occupied by the gather kernel

### B. Skewness Sensitivity

To assess the skewness [54] sensitivity of each scheduling scheme for given graph data, we compared SparseWeaver and $S_{vm}$, $S_{em}$ scheduling with various skewness patterns with the PageRank algorithm. Each graph data is generated with a fixed-sized edge (1.9M) and different vertex numbers (10k,

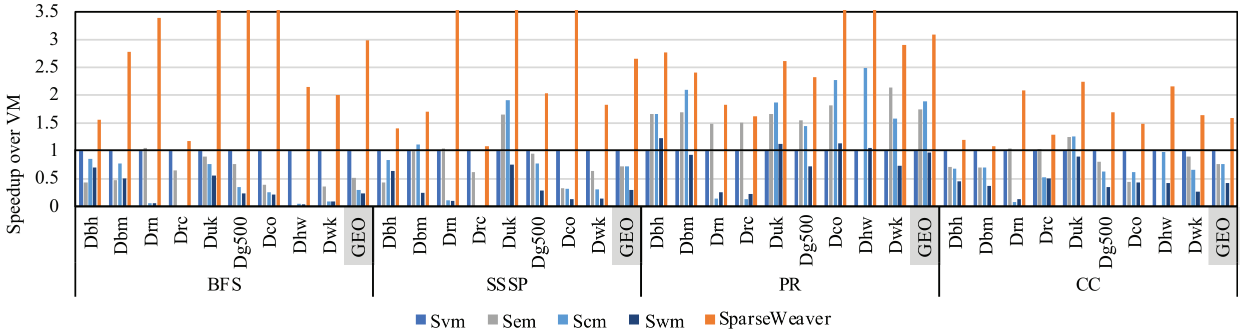
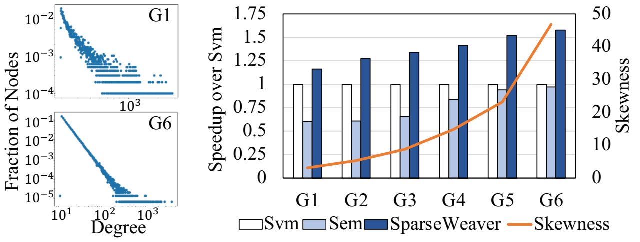


Fig. 10: Performance comparison with the four software scheduling schemes ($S_{vm}$, $S_{em}$, $S_{wm}$, $S_{cm}$) and SparseWeaver. Each graph shows speedups over $S_{vm}$. Four graph algorithms on nine graphs are used on Vortex GPU



(a) Fraction of Nodes

(b) Speedup over $S_{vm}$

Fig. 11: Skewness Sensitivity (a) shows the graph degree distribution of G1 (low skewness) and G6 (high skewness) (b) shows speedups over $S_{vm}$ when increasing skewness

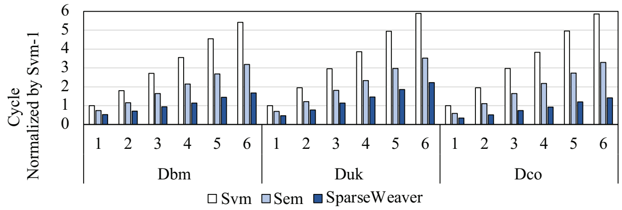


Fig. 12: Execution cycle comparison by increasing memory and GPU cycle ratio normalized by $S_{vm} - 1$ ratio. The number n means GPU has n times higher frequency compared to DRAM frequency (n GHz GPU versus 1 GHz DRAM)

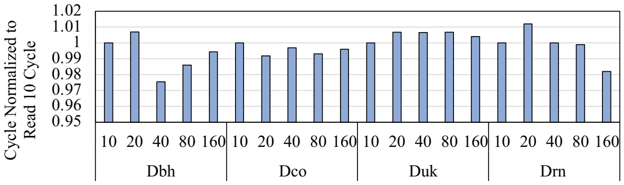


Fig. 13: Execution cycle versus cache read overhead for 10, 20, 40, 80, and 160 cycles.

12k, 16k, 20k, 40k, 80k) by the NetworkX Power-law graph generator. Figure 11a shows the degree distribution of G1 and G6 graphs. G1 has a small number of vertices and exhibits lower skewness, so it has a narrow degree distribution range and a short edge fraction tail. On the other hand, G6 has a large number of vertices, high skewness, a wide degree distribution, and a long edge fraction tail. As the graph index increases from G1 to G6, skewness also increases, which leads to progressively worse imbalances. The effect of skewness is shown in comparison with $S_{em}$ and $S_{vm}$ in Figure 11b, where both performances become similar as the workload imbalance increases even though EM reads double memory for edges compared to $S_{vm}$. On the contrary, SparseWeaver shows a similar increase trend as $S_{em}$, implying that workloads are balanced, leading to tolerance to the data skewness.

## C. *Effect of Memory Configuration*

Figure 12 illustrates that the cycle count increases linearly with the ratio of memory and GPU core frequency from 1 to 6 using $S_{vm}$, $S_{em}$, and SparseWeaver with the PageRank algorithm. The number n means GPU has n times higher frequency compared to DRAM frequency. This graph demonstrates that graph processing is a memory-intensive application. SparseWeaver shows better performance than $S_{vm}$ and $S_{em}$ across all cases because SparseWeaver reduces the total cycle count effectively by addressing workload imbalance with less memory access.

## D. *Effect of Work Table Access*

Figure 13 demonstrates the effect of memory access by the Sparse Workload Information table and Dense Work ID table by changing shared memory read overhead from 10, 20, 40, 80, and 160. SparseWeaver demonstrates that shared memory read latency can be concealed by the GPU pipeline. We tested shared memory read latency under varying conditions using an 8-core, 32-warp, and 32-thread Vortex configuration with the PageRank algorithm. With only one scan required for each table entry, performance remained consistent despite increases in latency, indicating that other instructions can effectively mask the cache read latency.

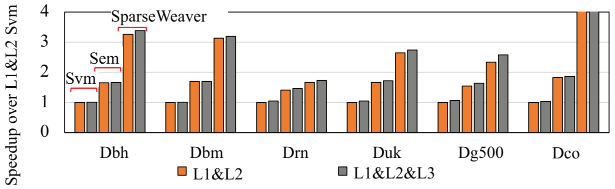


Fig. 14: Performance comparison with L1&L2 and L1&L2&L3 cache with PageRank. Each graph shows speedups over $S_{vm}$ with L1&L2 cache.

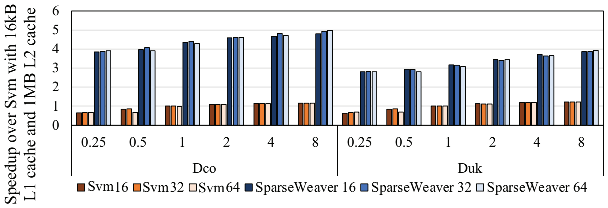


Fig. 15: Performance comparison by L1 cache size (16KB, 32KB, 64KB) and L2 cache size (0.25MB, 0.5MB, 1MB, 2MB, 4MB, 8MB) using PageRank and two datasets. Each graph shows speedups over $S_{vm}$ with 16KB L1 and 1MB L2 cache.

### E. Cache and Memory Analysis

Figure 14 shows the performance comparison between two cases ( L1&L2 cache, L1&L2&L3 cache). The graph shows that the existence of an L2 cache affects performance. The graph shows that the existence of an L3 cache has no significant impact when comparing the results of the L1&L2 case with the L1&L2&L3 case. Figure 15 shows the performance comparison by increasing the L1 from 16KB, 32KB, 64KB, and L2 cache size from 0.25MB, 0.5MB, 1MB, 2MB, 4MB, and 8MB. The graph also shows that cache size has no significant performance impact.

### F. Hardware Overhead

To assess the area overhead incurred in hardware, SparseWeaver was modeled in RTL for the Vortex GPU. The RTL was synthesized using Quartus Prime Pro 18.1 and targeted the Intel Stratix 10 FPGA. The Workload Info Table and Work ID Table logic (Figure 7) required an increase of 678 dedicated logic registers for a single core. The SparseWeaver FSM and other supporting logic for adding the new instructions required 3109 more adaptive logic modules (ALMs) for a single core. The overall per-core increase is 0.045% for dedicated logic registers and 2.96% for ALMs due to the Sparse Workload info Table, Dense Work ID table, and FSM. In the 16-core case, the ALM increase is 2.01% over the default 16-core configuration. This signifies a very small area overhead in hardware since SparseWeaver does not increase the usage of block memory, RAM blocks, or DSP blocks. Table IV gives a summary of the FPGA resources used. Figure 16 provides a visual representation of the increase in area for a 1-core and a 16-core GPU.

| Vortex GPU configuration | Total ALMs | ALM % increase | Block memory % increase | RAM % increase | DSP % increase |
|---|---|---|---|---|---|
| 1-core default | 105,094 | 2.96% | 0% | 0% | 0% |
| 1-core w/ SparseWeaver | 108,203 | | | | |
| 16-core default | 580,332 | 2.01% | 0% | 0% | 0% |
| 16-core w/ SparseWeaver | 591,971 | | | | |

TABLE IV: The area overhead of SparseWeaver in terms of FPGA resources utilized.

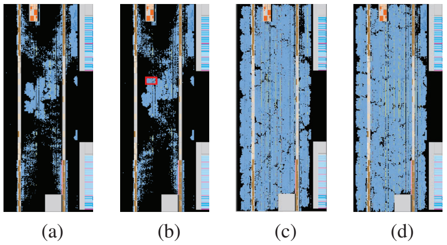

(a) (b) (c) (d)

Fig. 16: Block utilization diagrams for different configurations of the Vortex GPU synthesized on the Stratix 10 FPGA where blue indicates a used block and the red box indicates a major difference in the utilized blocks. (a) A single core of the GPU (b) A single core of the GPU with SparseWeaver (c) A 16-core GPU (d) A 16-core GPU with SparseWeaver

As for developmental overhead, the hardware implementation consists of an additional 251 lines of System Verilog code. Compared to the original codebase with 184,449 lines of System Verilog code, this represents a 0.136% increase.

### G. Push and Pull Breakdown

Figure 17 shows the performance breakdown with the Push and Pull direction. For registration, both Push and Pull incur a similar amount (less than 1% difference) of cycles. The summation cycles of the Edge schedule and Edge Info access for Push and Pull are also similar because we use a symmetric graph dataset. The gather and sum cycle varies by dataset. For example, Push shows fewer cycles with $D_{wk}$, $D_{rc}$, and Pull shows fewer cycles with other datasets.

### H. Case Study 1: Existing Hardware-based Scheme

In this case study, we compare SparseWeaver with edge-generating hardware (EGHW) mode similar to existing hardware-schemes [32], [42], [43]. The edge-generating hardware performs the entire operations in the edge schedule (blue box) except filtering the vertex in Figure 5, including investigating the graph topology of the vertex and edge information access. In EGHW mode, the GPU stores the vertex ID in a buffer in shared memory, and EGHW accesses graph topology information by reading the vertex ID in the buffer. Then, EGHW performs remapping and stores edge data, such as opposite vertex ID and weight, into the buffer. Therefore, GPU has to wait for edge information from EGHW to perform gather and sum. By moving graph topology and edge information access to Weaver, SparseWeaver can generate edge information using hardware.

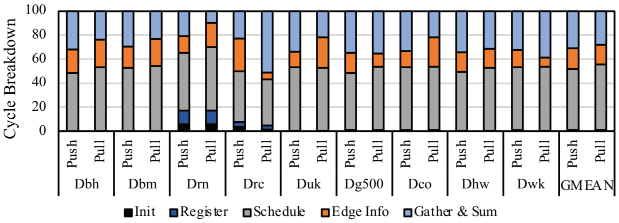


Fig. 17: Execution cycle breakdown of the gather process with Push and Pull using PageRank. The breakdown includes five steps: Init, Registration, Work ID calculation, Edge information access, and Gather & Sum.

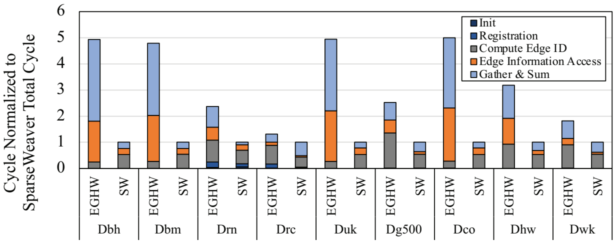


Fig. 18: Execution cycle breakdown of gathering process with EGHW and SparseWeaver using PageRank algorithms. Execution time is normalized to the total cycle of SparseWeaver.

Figure 18 shows execution time comparsion with EGHW and Weaver. SparseWeaver achieved a geometric mean speedup of 3.64 times compared to EGHW. In detail, the speedup comes from the distribution stage, such as Work ID calculation, Edge Information Access, and Gathering. The performance difference comes from the fact that EGHW does not conceal the overhead of edge information read from memory and needs more shared memory access to store and read generated edge information to the buffer. In addition, the stall caused by Edge Information Access also affected the Gather because of dependency on edge information load. It is possible to add a pipeline and more complex logic into EGHW to improve the performance of Edge Information Access, but we can achieve efficient memory access using the GPU pipeline like SparseWeaver design.

### I. Case Study 2: Performance with GCN

This case study demonstrates the extensibility of SparseWeaver by comparing the performance of the graph convolution network (GCN) operator [25] using vertex and weight parallelization strategies. The GCN experiment evaluates three kernels: initialization, sparse matrix-matrix multiplication (SpMM), and mean aggregation (GraphSum) across 16 weight dimension sizes. As a baseline, we change $S_{vm}$ mapping to first parallelize the weight dimension and the vertex dimension to execute the SpMM and GraphSum kernels, so each thread gathers a specific weight across the size of the vertex's neighbor list and can remove atomic for weight update. On the other hand, our method continues to parallelize edge updates by iterating through the weight dimension using atomic operation.

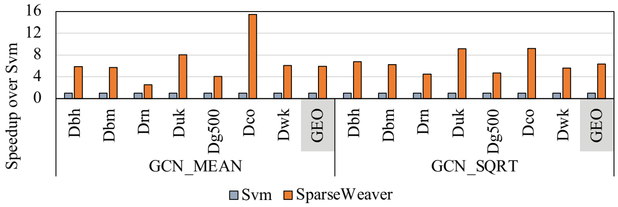


Fig. 19: Performance of GCN operators with three different scheduleing methods, $S_{vm}$ parallelized by weight.

Figure 19 indicates that SparseWeaver achieves a 6.15 times speedup compared to $S_{vm}$. SpMM demonstrates improved performance with $S_{vm}$ due to the benefits of weight parallelization and reduced use of atomic operations, while for GraphSum, SparseWeaver performs better by reducing the cost of coefficient calculations, determined by the degree of the source and destination vertices. Given that GraphSum requires more processing time, SparseWeaver shows better performance than $S_{vm}$.

### J. Case Study 3: Comparison with Existing Auto-tuner

| | Autotuner with A30<br>(108 SM, 64 warps/core, 1200Mhz) | | | | SparseWeaver<br>(8 cores, 32 warps/core) | | |
|---|---|---|---|---|---|---|---|
| | Tuning Time (sec) | $S_{vm}$ (ms) | Best (ms) | Speedup | $S_{vm}$ (ms) | SW (ms) | Speedup |
| $D_{hw}$ | 4502.83 | 18.92 | 8.78 | 2.16 | 5408.33 | 1016.67 | 5.32 |
| $D_{uk}$ | 1446.57 | 1.45 | 0.69 | 2.11 | 400.83 | 173.33 | 2.61 |
| $D_{co}$ | 1710.66 | 2.63 | 1.40 | 1.88 | 916.67 | 216.67 | 4.23 |
| $D_{rn}$ | 1139.41 | 0.47 | 0.32 | 1.47 | 453.01 | 247.68 | 1.83 |

TABLE V: Speedup over $S_{vm}$ Comparison using Autotuner [21] and SparseWeaver with PageRank

Table V shows the performance gain compared to $S_{vm}$ with the existing Autotuner [21] and SparseWeaver. Even though hardware configurations, such as the number of parallel units, differ between the tested Nvidia GPU and SparseWeaver setup, the table shows that SparseWeaver has better performance compared to $S_{vm}$, even without requiring the tuning time that the Autotuner demands.

## VI. Related Work

### A. Software workload balancing method

Some existing works [6], [12], [20], [21], [24], [31], [33], [34], [37] suggest software-based schemes to address workload imbalance problem. Some software-based scheduling schemes [6], [12], [33], [34] store the graph topology of the target vertex range in shared or global memory and employ binary search or atomic operations with synchronization methods to determine which edges to process. For example, Warp Mapping ($S_{wm}$) and Cooperative Thread Array (CTA) Mapping [33] assign a vertex and its neighboring edges to a warp and a CTA, respectively. $S_{wm}$ and CTA employ shared memory to save neighbor list information and use a binary search to find an edge to process. Some works [6], [34] also propose more complex scheduling, which uses multiple buckets to classify neighbor lists into small, medium, and large categories to improve reusability and reduce search costs. However, because of the hardness of convincing which scheduling performs better for specific graphs, some existing [6], [21], [33] works suggest auto-tuners to find the optimal solutions. Other software-based schemes change storage formats [20], [24], [31], [37], but changing requires static time efforts such as vertex virtualization [31], [37], subgraphing [20].

### B. Hardware workload balancing method

Some hardware-based workload balancing schemes [32], [42], [43] propose hardware accelerators that generate active edge information. SCU [42], [43] suggests special hardware outside of GPU which reads graph topology and edge information, filters edges, generates compact edge frontier, and stores edge frontier in global memory. GraphPEG [32] proposes a special hardware per core that reads graph topology and edge information stored in global memory and stores edge information in shared memory. Both methods need additional memory usage to store generated edge information, and special hardware generates load and store requests for edge access inside special hardware by itself. Both papers aim to enhance GPGPU efficiency for graph processing by addressing irregular memory accesses, offering significant performance and energy benefits despite their limited applicability or scalability.

There are several frameworks [17], [39], [53] that focus on load balancing strategies on FPGA platforms. [39] addresses the severe workload imbalance in Graph Attention Networks (GATs) caused by the power-law distribution of real-world graph data by selecting only the top-K neighbor nodes with the highest attention scores for the aggregation phase. [53] focuses on optimizing the inference of Graph Convolutional Networks (GCNs) on FPGAs by managing data flow and reducing redundant memory accesses, thus enhancing performance by leveraging the parallel processing capabilities of FPGAs while balancing memory access loads. Meanwhile, [17] tackles the irregularities in data access and control flow due to the power-law distribution in graph data by employing dynamic auto-tuning techniques. Limitations of the techniques are their applicability and system complexity, where SparseWeaver decouples algorithm and load balancing, offering more flexibility in its application.

## VII. Discussion

### A. General Usage of SparseWeaver

We believe that SparseWeaver can extend its applicability to other sparse applications, particularly those originally using the CSR format, such as GPU hashing, MapReduce, Graph Neural Networks, or sparse matrix multiplication. These applications handle sparse data, such as hash tables containing sparse workload information. For example, Algorithm 1 shows a possible implementation of GPU hash lookup. The sparse workload information can store the position of each key-value pair within hash table buckets [2]. Sparse-Weaver can replace the second for loop (Line 3, which uses sparse information in offset array) to distribute hash operations across multiple threads, as the offset array contains workload information.

**Algorithm 1:** The GPU hash lookup [2]

```
  Input: keys : Input Hash Keys
         offset: offset array pointing to the ranges of bucket
1 foreach $key \in keys$ do
2     $bucket \leftarrow hash(key)$
3     for $i \in range(offset[bucket], offset[bucket+1])$ do
4         if $hashtable[i] == key$ then
5             ...
6     end
7 end
```

### B. Integrating SparseWeaver into Auto-tuners

Integrating SparseWeaver into existing auto-tuners can be achieved by extending hardware options. Auto-tuners often struggle to incorporate new hardware features, such as tensor or sparse cores, without modifying code because GPU architectures, drivers, and programming languages evolve rapidly. Therefore, supporting these new hardware features requires extensions in auto-tuners. SparseWeaver represents a new microarchitectural feature that often outperforms software-based schedules for diverse graphs. Consequently, if the auto-tuner can run on GPUs supporting SparseWeaver, it could be extended with the hardware option SparseWeaver. This approach may eliminate the need to explore software schedules for supported GPUs while preserving the use of software schedules for GPUs that lack SparseWeaver support.

## VIII. Conclusion

This work proposes a new collaborative hardware and software graph processing framework SparseWeaver. SparseWeaver effectively addresses the workload imbalance in the graph processing on GPU by converting the sparse operations into dense operations. Based on the analysis of common patterns in software schemes, we propose a hardware logic, called Weaver, new lightweight hardware that is tightly integrated into the GPU pipeline with simple ISA. With only 0.045% additional dedicated logic registers and 2.96% additional ALMs in a single core, SparseWeaver achieves a performance speedup that is 2.36 times faster on real-world graph datasets compared with the software scheme.

## Acknowledgements

This research was partially supported by NSF PPOSS 2119523, Intel, AMD, and BAH, Samsung, No. RS-2022-II220050 funded by the Ministry of Science and ICT, and BK21 from South Korea Government. We also want to acknowledge the research infrastructure and services provided by the Rogues Gallery testbed hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under NSF Award Number #2016701. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

## References

[1] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, “Heterogeneous memory subsystem for natural graph analytics,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 134–145.

[2] D. A. F. Alcantara, “Efficient hash tables on the gpu,” Ph.D. dissertation, USA, 2011.

[3] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, “Groute: An asynchronous multi-gpu programming model for irregular computations,” *ACM SIGPLAN Notices*, vol. 52, pp. 235–248, 2017.

[4] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To push or to pull: On reducing communication and synchronization in graph computations,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[5] P. Boldi and S. Vigna, “The webgraph framework ii: codes for the world-wide web,” in *Data Compression Conference, 2004. Proceedings. DCC 2004*, 2004, pp. 528–.

[6] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, “Compiling graph applications for gpu s with graphit,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 248–261.

[7] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer Networks and ISDN Systems*, vol. 30, pp. 107–117, 1998, proceedings of the Seventh International World Wide Web Conference.

[8] S. Brin and L. Page, “Reprint of: The anatomy of a large-scale hypertextual web search engine,” *Computer networks*, vol. 56, pp. 3825–3833, 2012.

[9] X. Chen and L. Pan, “A survey of graph cuts/graph search based medical image segmentation,” *IEEE Reviews in Biomedical Engineering*, vol. 11, pp. 112–124, 2018.

[10] B. Coleman, S. Segarra, A. J. Smola, and A. Shrivastava, “Graph reordering for cache-efficient near neighbor search,” in *Advances in Neural Information Processing Systems*, vol. 35. Curran Associates, Inc., 2022, pp. 38 488–38 500.

[11] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, “Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 752–768.

[12] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-efficient parallel gpu methods for single-source shortest paths,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 349–359.

[13] G. Donnelly, “75 super-useful facebook statistics for 2018,” https://www.wordstream.com/blog/ws/2017/11/07/facebook-statistics, accessed: 2024-06-25.

[14] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. kim, “Vortex: OpenCL Compatible RISC-V GPGPU,” Feb. 2020.

[15] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 251–262.

[16] 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.

[17] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, “Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 922–936.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed Graph-Parallel computation on natural graphs,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[19] V. GPGPU, “Vortex: A RISC-V GPGPU,” https://github.com/vortexgpgpu/vortex, accessed: 2024-04-18.

[20] S. Jeong, S. Cho, Y. Lee, H. Park, S. Heo, G. Kim, Y. Kim, and H. Kim, “Cr2: Community-aware compressed regular representation for graph processing on a gpu,” in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024.

[21] S. Jeong, Y. Lee, J. Lee, H. Choi, S. Song, J. Lee, Y. Kim, and H. Kim, “Decoupling schedule, topology layout, and algorithm to easily enlarge the tuning space of gpu graph processing,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2023, p. 198–210.

[22] P. Jääskeläinen, C. Sánchez de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, “pocl: A performance-portable opencl implementation,” *International Journal of Parallel Programming*, 2015.

[23] 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)*, 2015, pp. 39–50.

[24] 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*, 2014, pp. 239–252.

[25] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[26] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi:large-scale graph computation on just a pc,” in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[27] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, “Gpop: A cache and memory-efficient framework for graph processing over partitions,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 393–394.

[28] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” ser. CGO ’04. IEEE Computer Society, 2004, p. 75.

[29] H. Liu and H. H. Huang, “Enterprise: breadth-first graph traversal on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[30] H. Liu and H. H. Huang, “Simd-x: Programming and processing of graph algorithms on gpus,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019, pp. 411–427.

[31] W. Liu and B. Vinter, “Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[32] Y. Lü, H. Guo, L. Huang, Q. Yu, L. Shen, N. Xiao, and Z. Wang, “Graphpeg: Accelerating graph processing on gpus,” *ACM Trans. Archit. Code Optim.*, vol. 18, 2021.

[33] K. Meng, J. Li, G. Tan, and N. Sun, “A pattern based algorithmic autotuner for graph processing on gpus,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 201–213.

[34] D. Merrill, M. Garland, and A. Grimshaw, “Scalable gpu graph traversal,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[35] D. Merrill, M. Garland, and A. Grimshaw, “High-performance and scalable gpu graph traversal,” *ACM Trans. Parallel Comput.*, vol. 1, 2015.

[36] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, “Exploiting locality in graph analytics through hardware-accelerated traversal scheduling,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[37] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, *Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing*, 2018, pp. 622–636.

[38] S. Pai and K. Pingali, “A compiler for throughput optimization of graph algorithms on gpus,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[39] N. Park, D. Ahn, and J.-J. Kim, “Workload-balanced graph attention network accelerator with top-k aggregation candidates,” in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[40] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[41] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proceedings of the*

*Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[42] A. Segura, J.-M. Arnau, and A. González, "Scu: A gpu stream compaction unit for graph processing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 424–435.

[43] A. Segura, J.-M. Arnau, and A. González, "Irregular accesses reorder unit: improving gpgpu memory coalescing for graph-based workloads," in *The Journal of Supercomputing*, 2022, pp. 762–787.

[44] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," jan 2018.

[45] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[46] A. t. Balaban, "Applications of graph theory in chemistry," *Journal of chemical information and computer sciences*, 1985.

[47] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper, and H. Kim, "Skybox: Open-source graphic rendering on programmable risc-v gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[48] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[49] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sepgraph: Finding shortest execution paths for graph processing under a hybrid framework on gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 38–52.

[50] 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.

[51] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[52] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[53] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 29–39.

[54] D. Zwillinger and S. Kokoska, *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.