WER: Maximizing Parallelism of Irregular Graph Applications Through GPU Warp EqualizeR

En-Ming Huang[†], Bo-Wun Cheng[†], Meng-Hsien Lin[‡], Chun-Yi Lee[†], and Tsung Tai Yeh[‡]
[†]Elsa Lab, Department of Computer Science, National Tsing Hua University, Hsinchu City, Taiwan
[‡]Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu City, Taiwan

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473955

Abstract

Irregular graphs are becoming increasingly prevalent across a broad spectrum of data analysis applications. Despite their versatility, the inherent complexity and irregularity of these graphs often result in the underutilization of Single Instruction, Multiple Data (SIMD) resources when processed on Graphics Processing Units (GPUs). This underutilization originates from two primary issues: the occurrence of inactive threads and intra-warp load imbalances. These issues can produce idle threads, lead to inefficient usage of SIMD resources, consequently hamper throughput, and increase program execution time. To address these challenges, we introduce **W**arp **E**qualize**R** (WER), a framework designed to optimize the utilization of SIMD resources on a GPU for processing irregular graphs. WER employs both software API and a specifically-tailored hardware microarchitecture. Such a synergistic approach enables workload redistribution in irregular graphs, which allows WER to enhance SIMD lane utilization and further harness the SIMD resources within a GPU. Our experimental results over seven different graph applications indicate that WER yields a geometric mean speedup of 2.52× and 1.47× over the baseline GPU and existing state-of-the-art methodologies, respectively.

I. Introduction

Graph analytics offers a means of discovering key insights from vast volumes of highly interconnected data and has been extensively integrated into various fields, including social networks [1], medical sciences [2], and cryptocurrency transactions [3]. A graph is typically defined as a structure that consists of vertices and edges, and is employed predominantly for interpreting the relationships between distinct data components. In the context of large-scale graph processing, programmable General-Purpose Graphics Processing Units (GPGPUs) are often considered favorable due to their ability to execute graph workloads in parallel. Nevertheless, mapping vertices and edges of a graph onto the Single Instruction Multiple Thread (SIMT) execution units of GPGPUs, while fully exploiting the potential of parallel cores, presents a formidable challenge. This difficulty arises as the quantity of edges associated with each vertex in a graph often exhibits variability in real-world graph workloads. Such irregular graph applications often lead to inefficiencies when paired with the uniform Single Instruction Multiple Data (SIMD) execution units, primarily due to *inactive threads* and *intra-warp load imbalance* issues. These challenges, as demonstrated in Fig. 1, necessitate prompt and effective solutions.

Fig. 1 (a) presents an example execution order of the Breadth-First Search (BFS) algorithm, where the arrows in

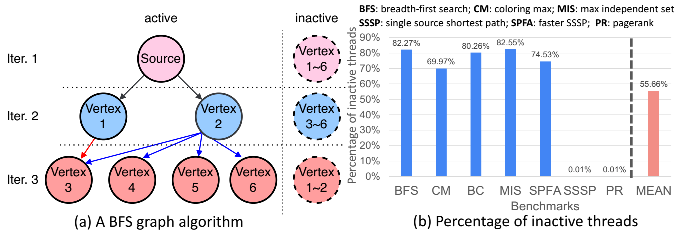


Fig. 1. Challenges in processing irregular graphs on GPUs: (a) A breadth first search (BFS) graph algorithm for highlighting the *inactive threads* and *intra-warp load imbalance* issues, and (b) a percentage breakdown of inactive threads in seven graph algorithms.

the figure represent the directed edges of the graph. In each iteration, vertices are classified into two types, activated and inactive. Only activated vertices perform calculations on their neighbors (e.g., marking the vertex as 'traveled' in BFS). It can be observed that the degree of parallelism is dependent on the number of active threads, which can vary across iterations. Therefore, mapping vertices onto GPU threads can lead to the *inactive threads* issue, and results in underutilization of SIMD lanes. Fig. 1 (b) serves as a motivational experiment that compares the percentage of inactive threads in several representative graph algorithms [4, 5]. The results reveal that approximately 70% or more of threads are inactive in the majority of these graph algorithms. Such control flow inefficiency reduces the SIMD lane utilization and the overall performance.

On the other hand, Fig. 1 (a) also emphasizes the issue of *intra-warp load imbalance*. In the SIMT execution model, where the threads in a warp operate in lock-step, any load imbalance can lead to idle SIMD lanes and prolong the completion time of the warp. Unlike the *inactive threads* problem which is dependent on the algorithm, the *intra-warp load imbalance* issue represents a critical challenge posed by irregular graphs. This is due to the significant variance in the number of edges associated with each vertex. This problem is comparatively more noticeable than the *inactive threads* issue and has therefore been attempted in several state-of-the-art (SOTA) works. Previous SOTA software approaches, such as collaborative task engagement (CTE) [6, 7] and warp-centric [8], have adopted an edge-based processing scheme as depicted in Fig. 2. This scheme processes the workloads of a single vertex simultaneously by the threads within a warp. In contrast, Tigr [9] aimed to adapt the data layout of irregular graphs to a SIMD execution-friendly format by limiting the maximum degree of each vertex. GraphPEG [10] and SCU [11], on the other hand, proposed custom hardware units in conjunction with the GPU to balance workloads. However, these hardware approaches demand additional memory space, and none of them mentioned above address the *inactive threads* issue.

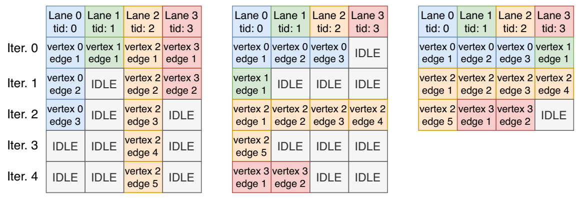
To address the unresolved issues from previous endeavors, this paper introduces **W**arp **E**qualize**R** (WER), a synergistic software and hardware framework specifically designed to tackle the challenges of *intra-warp load imbalance* and *inactive threads*. WER comprises the development of a software programming API, which incorporates two custom intrinsics to enable communication between programs and the WER microarchitecture. In addition, it includes the integration of a custom hardware microarchitecture and control flow into a GPU pipeline. This setup is designed to synergize with the WER programming API. Through this holistic approach, the workloads of active threads within a warp are redistributed across the active SIMD lanes, effectively addressing the *intra-warp load imbalance* issue. Moreover, we extend WER into WER+ by modifying the SIMT stack to allow all inactive threads to contribute to the computation. This enables tackling the performance bottleneck caused by *inactive threads*. As a result, WER+ achieves a geometric mean speedup of 2.52× and 1.47× over the baseline GPU and the most effective SOTA method across seven representative graph algorithms on five real-world graph datasets. WER also significantly reduces the average energy consumption by over 50%. The contributions of this work are as follows.

- Identification and thorough examination of the inactive threads and the intra-warp load imbalance issues.
- Development of a synergistic software-hardware framework called WER. This framework includes a programming API and a custom hardware microarchitecture designed to address intra-warp load imbalance issue.
- Extension of WER to WER+ to harness the inactive threads for enhancing the overall SIMD efficiency.
- Validation with extensive experiments across various representative graph algorithms and real-world datasets for demonstrating the effectiveness and practicality of the proposed WER and WER+ frameworks.

The paper is organized as follows. Section II highlights the challenges and the motivation of this work, Section III elaborates on the proposed framework, Section IV presents the experimental results. Section V concludes the paper.

II. Challenges of Processing Irregular Graphs

In this section, we present the problem specification and discuss the motivation of this work. As outlined in Section I, a number of contemporary graph-based algorithms demand multiple rounds of iterative computations to reach convergence. To enact parallel processing on a GPU, previous endeavors typically utilize either of two different assignment strategies for distributing tasks to the SIMD lanes: (1) a vertex-based scheme, and (2) an edge-based scheme, as depicted in Fig. 2 (a) and Fig. 2 (b), respectively. The former allocates each vertex to a specific GPU thread (i.e., a SIMD lane) for processing, while the latter assigns edges connected to the same vertex to be processed within a single warp. Unfortunately, these two schemes face significant challenges from the issue of idle threads. These idle threads arises from from the disparity in the number of edges connected to different vertices, a



	Lane 0 tid: 0	Lane 1 tid: 1	Lane 2 tid: 2	Lane 3 tid: 3
Iter. 0	vertex 0 edge 1	vertex 1 edge 1	vertex 2 edge 1	vertex 3 edge 1
Iter. 1	vertex 0 edge 2	IDLE	vertex 2 edge 2	vertex 3 edge 2
Iter. 2	vertex 0 edge 3	IDLE	vertex 2 edge 3	IDLE
Iter. 3	IDLE	IDLE	vertex 2 edge 4	IDLE
Iter. 4	IDLE	IDLE	vertex 2 edge 5	IDLE

(a) Vertex-based scheme.

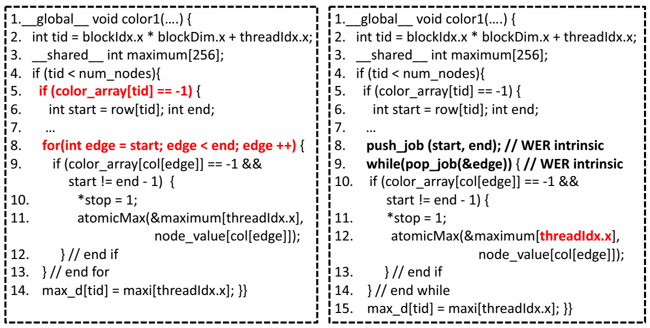
Lane 0 tid: 0	Lane 1 tid: 1	Lane 2 tid: 2	Lane 3 tid: 3
vertex 0 edge 1	vertex 0 edge 2	vertex 0 edge 3	IDLE
vertex 1 edge 1	IDLE	IDLE	IDLE
vertex 2 edge 1	vertex 2 edge 2	vertex 2 edge 3	vertex 2 edge 4
vertex 2 edge 5	IDLE	IDLE	IDLE
vertex 3 edge 1	vertex 3 edge 2	IDLE	IDLE

(b) Edge-based scheme.

Lane 0 tid: 0	Lane 1 tid: 1	Lane 2 tid: 2	Lane 3 tid: 3
vertex 0 edge 1	vertex 0 edge 2	vertex 0 edge 3	vertex 1 edge 1
vertex 2 edge 1	vertex 2 edge 2	vertex 2 edge 3	vertex 2 edge 4
vertex 2 edge 5	vertex 3 edge 1	vertex 3 edge 2	IDLE

(c) WER.

Fig. 2. An illustrative comparison of task distribution schemes, highlighting the intra-warp load imbalance issue for vertex-based methods [9, 10], edge-based approaches [6, 7, 8], and our WER.



```
1.__global__ void color1(....) {
2.  int tid = blockIdx.x * blockDim.x + threadIdx.x;
3.  __shared__ int maximum[256];
4.  if (tid < num_nodes){
5.    if (color_array[tid] == -1) {
6.      int start = row[tid]; int end;
7.      ...
8.     for(int edge = start; edge < end; edge ++) {
9.       if (color_array[col[edge]] == -1 &&
             start != end - 1)  {
10.          *stop = 1;
11.          atomicMax(&maximum[threadIdx.x],
                                node_value[col[edge]]);
12.      } // end if
13.   } // end for
14.  max_d[tid] = maxi[threadIdx.x]; }}
```

(a) Color_Max kernel in vertex-based scheme

```
1.__global__ void color1(....) {
2.  int tid = blockIdx.x * blockDim.x + threadIdx.x;
3.  __shared__ int maximum[256];
4.  if (tid < num_nodes){
5.    if (color_array[tid] == -1) {
6.      int start = row[tid]; int end;
7.      ...
8.     push_job (start, end); // WER intrinsic
9.     while(pop_job(&edge)) { // WER intrinsic
10.     if (color_array[col[edge]] == -1 &&
            start != end - 1) {
11.         *stop = 1;
12.          atomicMax(&maximum[threadIdx.x],
                                node_value[col[edge]]);
13.       } // end if
14.   } // end while
15.   max_d[tid] = maxi[threadIdx.x]; }}
```

(b) Color_Max kernel with WER intrinsics

Fig. 3. An GPU kernel function vs. the one with WER intrinsics.

characteristic particularly prominent in irregular graphs. In the motivational example illustrated in Fig. 3 (a), if the number of edges associated with the vertices vary (e.g., line 8, where 'start' and 'end' denote the index range of the edges linked to a vertex), intra-warp load imbalance issue could arise. In addition to the above issue, problems stemming from inactive threads may further undermine GPU usage efficiency. Inactive threads typically result from branch conditions, and this inefficiency can inherently be caused by the algorithm itself. For instance, the coloring max kernel function in Fig. 3 (a) requires a branch condition (e.g., line 5) to choose the uncolored vertices, and therefore only a subset of vertices are selected to perform updates. In light of these challenges, the primary objective of this work is to formulate a WER framework that effectively overcomes these obstacles, and fully harnesses the SIMD resources in a GPU, as visualized in Fig. 2 (c).

III. WER: A Synergistic SW & HW Framework

To accomplish our objective, the WER framework integrates both software and hardware components. These include the introduction of new software programming API (i.e., Fig. 3 (b)), along with tailored hardware microarchitectures (i.e., Fig. 4). They are described as follows.

A. WER Programming API for Workload Reassignment

WER optimizes the utilization of SIMD lanes by integrating the benefits of vertex- and edge-based schemes. Initially, tasks are distributed in a vertex-based manner. WER pushes and logs the workloads of each vertex, corresponding to each SIMD lane, into an indexing table, as depicted in Fig. 4 (a). It then redistributes the workloads in an edge-based fashion by 'popping' tasks and redistributing them across the active SIMD lanes. This process

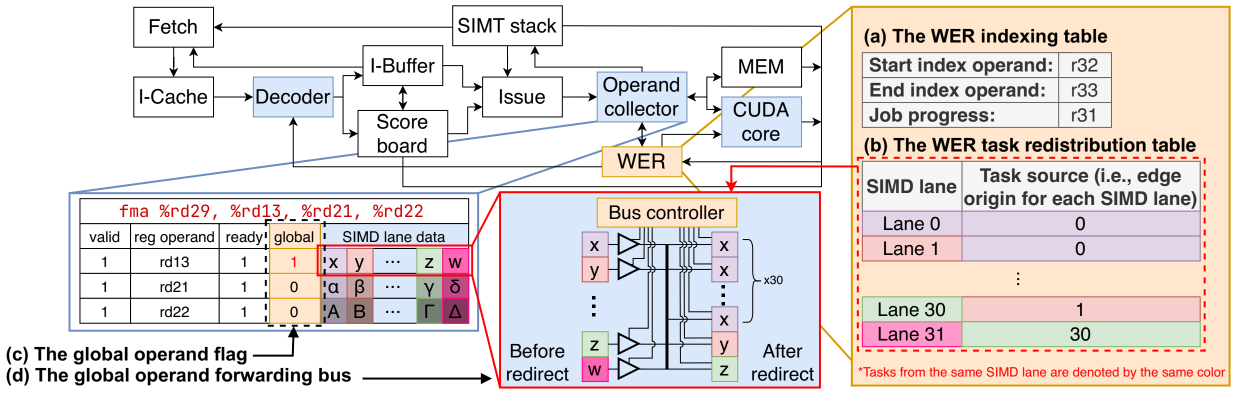


Fig. 4. The WER-enhanced GPU architecture. In this example, the operands `r32` and `r33` store the index bounds in the register file. Threads 0∼29 are processing the edges from lane 0, while the edges in lanes 1 and 30 are redistributed to threads 30 and 31, respectively.

addresses the intra-warp load imbalance issue by ensuring that the active SIMD lanes are all engaged with tasks.

The WER programming API, as illustrated in Fig. 3 (b), is composed of two custom intrinsics: **push_job** and **pop_job**. During compilation, the original 'for' loop structure that assigns a specific vertex to a GPU active thread (e.g., lines 8-13 of Fig. 3 (a)), is replaced by these custom intrinsics, which are then translated into the corresponding equivalent PTX assembly codes. The **push_job** intrinsic is responsible for recording the index bounds of edges for each active thread, while the **pop_job** intrinsic retrieves and redistributes workloads across the active SIMD lanes. When these intrinsics are encountered during execution, the WER microarchitecture logs the quantity of remaining workloads (i.e., edges) into the WER *indexing table* and reassigns them. This enables rescheduling of vertex tasks.

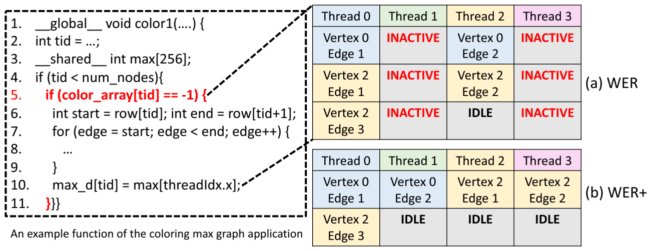
B. WER Microarchitecture for Workload Redistribution

Fig. 4 presents the WER microarchitecture and control flow integrated into the stream microprocessor (SM) pipeline in a GPU. This is designed to work in synergy with the WER programming API. The incorporated WER module interacts with three main components: the decoder, operand collector, and CUDA cores. This microarchitecture is designed to redistribute the workloads of active threads within a warp across the active SIMD lanes. WER examines the SIMT stack to determine the number of active threads in a warp. Redistribution of edges in WER occurs only when the decoder detects the assembly code associated with the custom intrinsics. In order to facilitate this process, several hardware tables and control unit are introduced: the *indexing table*, *task redistribution table*, *global operand flag*, and *global operand forwarding bus*.

Indexing table: In its implementation, the WER microarchitecture utilizes an indexing table to log and reference the operand that stores the 'start' and 'end' indices of edges associated with each vertex as conveyed by the **push_job** custom intrinsic. In the example illustrated in Fig. 4 (a), the 'start' and 'end' indices in the same warp are stored in operands `r32` and `r33`, respectively. An additional operand, `r31`, is employed to track job progress for different vertices. Each operand corresponds to a set of registers across the SIMD lanes, and is housed in the register file of the SM. Assuming the maximum number of operands is 2^{10}, we allocate a mere ten bits to represent each operand, and thus drastically reduce the storage overhead to just 30 bits per warp. Please note that the indexing table records only the operand numbers, obviating the need for creating a new one to store the index bounds.

Task redistribution table: WER employs a *task redistribution table* to facilitate workload redistribution among the active SIMD lanes within a warp, as shown in Fig. 4 (b). It records the source SIMD lane to which each task was initially assigned. However, simply distributing a lane's workload to other active lanes could lead to incorrect execution results due to potential differences in the necessary register values of variables. For example, in line 12 of Fig. 3 (b), `threadIdx.x` is referenced and expected to be stored in a register. If, due to workload redistribution, threads 0 and 1 are both executing thread 0's tasks, their `threadIdx.x` must both be set to 0 to guarantee program correctness. This represents a data dependency problem that necessitates the implementation of a register redirection mechanism. To deal with this challenge, the *task redistribution table* keeps track of the originating SIMD lane for each operand. By consulting the redistribution table, register values can be correctly forwarded from the originating lanes to the newly assigned lanes to ensure correct data referencing and consistent program execution.

Global operand flag: While the *task redistribution table* governs task distribution, the process of identifying and forwarding the necessary operands to different SIMD lanes remain crucial. In assembly instructions for graph processing, operands can be either global or local. Global operands represent data required across lanes during processing, while local operands pertain to data specific to each individual lane. The *global operand flag* in WER is adopted to identify the register operands that are global and hence should be forwarded to different SIMD lanes according to the aforementioned *task redistribution table* during edge processing. During compilation, the compiler identifies whether a operand is global and encodes this information into the instruction. Upon instruction decoding, the *global* attribute is dispatched to the operand collector's



```
1.  __global__ void color1(....) {
2.  int tid = ...;
3.  __shared__ int max[256];
4.  if (tid < num_nodes){
5.    if (color_array[tid] == -1) {
6.      int start = row[tid]; int end = row[tid+1];
7.      for (edge = start; edge < end; edge++) {
8.        ...
9.      }
10.     max_d[tid] = max[threadIdx.x];
11.   }}}
```

An example function of the coloring max graph application

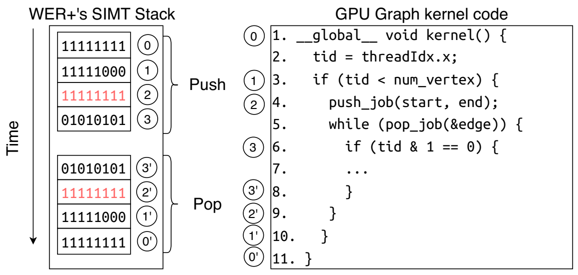
(a) WER

Thread 0	Thread 1	Thread 2	Thread 3
Vertex 0 Edge 1	INACTIVE	Vertex 0 Edge 2	INACTIVE
Vertex 2 Edge 1	INACTIVE	Vertex 2 Edge 2	INACTIVE
Vertex 2 Edge 3	INACTIVE	IDLE	INACTIVE

(b) WER+

Thread 0	Thread 1	Thread 2	Thread 3
Vertex 0 Edge 1	Vertex 0 Edge 2	Vertex 2 Edge 1	Vertex 2 Edge 2
Vertex 2 Edge 3	IDLE	IDLE	IDLE

Fig. 5. WER vs WER+: Tackling the inactive threads problem.



```
1. __global__ void kernel() {
2.   tid = threadIdx.x;
3.   if (tid < num_vertex) {
4.     push_job(start, end);
5.     while (pop_job(&edge)) {
6.       if (tid & 1 == 0) {
7.       ...
8.       }
9.     }
10.  }
11. }
```

Fig. 6. A detailed illustration of WER+'s SIMT stack operations.

global indicator field. If an operand is a global variable, the *global operand forwarding bus* work in conjunction with the *task redistribution table* to forward operand values to ensure data dependency across different SIMD lanes.

Fig. 4 (c) illustrates an example implementation of the *global operand flag*, represented as the *global* column within the operand collector. The operand collector in a GPU is tasked with gathering the source operands from the register file for instruction execution. In the provided example, the fused multiply-add (fma) instruction (`a*b+c`) multiplies operands `rd13` (a global operand) and `rd21` (a local operand). The result is then incremented by `rd22` and stored in `rd29`. In our design, destination registers (e.g., `rd29`) are identified as local operands and are not forwarded, because the computed result is written back to the shared or global memory in common graph applications, as line 12 in Fig. 3(b) shows. Given that a GPU's instruction set (which is equivalent to NVIDIA's PTX) typically contains at most three source operands, WER designates three bits within the PTX instructions to indicate the global flags, as depicted in Fig. 4 (c). It is important to note that encoding the *global operand flag* into instructions does not lead to additional hardware overhead, owing to the abundance of unused bits in the PTX instruction set, as revealed by the reverse engineering endeavors in [12].

Global operand forwarding bus: Upon identifying the global operands to be forwarded and their sources, these operands are transported to different SIMD lanes by the *global operand forwarding bus*, as depicted in Fig. 4 (d). The *global operand forwarding bus* fulfills two primary functions: (1) selecting the SIMD lanes to which the global operand should be forwarded by referring to the *task redistribution table*, and (2) broadcasting the global operand to the relevant SIMD lanes. This bus is governed by a bus controller, which uses the *task redistribution table* to arbitrate access to the shared bus. The number of cycles required to broadcast is determined by the number of distinct source lanes presented in the *task redistribution table*. For instance, WER incurs a three-cycle delay to forward data when there are three task sources (0, 1, and 30). Although it requires a maximum of 32 cycles in the worst-case scenario (where all 32 lanes are assigned workload from 32 different originating lanes), this design still offers substantial hardware efficiency compared to alternatives such as crossbars or dedicated networks. According to the experimental results discussed in Section IV.D, the latency overhead associated with the proposed design is minimal.

C. WER+: Utilizing the Potential of Inactive SIMD Lanes

While WER effectively redistributes edges among the active SIMD lanes to balance the workload, inactive threads can still exist. These inactive threads potentially reduce parallelism and limit the efficiency of WER's edge redistribution. As illustrated in Fig 5, divergent control flows resulting from the branch condition in line 5 can lead to inactive threads within a warp. The WER approach, discussed in Section III.B, does not utilize these inactive threads. To address this, we further introduce WER+, an enhanced microarchitecture that adjusts the mask of the SIMT stack to harness inactive threads. As depicted in Fig. 5 (b), WER+ offers the potential to further optimize parallelism within a warp for irregular graph applications.

To illutrate the functionality of WER+, Fig. 6 presents an example of a graph processing kernel and the corresponding status of the SIMT stack when utilizing WER+. The SIMT stack is a feature of a GPU that maintains the active mask of threads within a warp. The top-of-stack entry is updated or removed as the program's control flow progresses, particularly when it reaches reconvergence points or other synchronization events. This mask serves to indicate the divergence of each thread due to a branch condition. In our example, lines 1-2 initiate the process by pushing an entry onto the SIMT stack that sets all SIMD lanes as active. Upon reaching a branch condition at line 3, threads diverge to different control flow paths based on the comparison statement. Consequently, only a subset of SIMD lanes remains active, which limits WER's capability to use inactive threads for processing a vertex's edges.

To address this limitation and maximize the utilization of available SIMD lanes, WER+ introduces an enhanced **push_job** instruction that sets an active mask for all lanes. In practice, the **push_job** instruction in WER+ injects an additional mask that activates all entries. The inserted activation mask enables GPU to fully utilize all SIMD lanes for workload redistribution, without being confined to active lanes only. This technique activates inactive SIMD lanes during the processing phase. Once the region determined by the last **pop_job** is exited, the mask is restored to its previous state. It is important to note that the forwarding mechanism described in Section III.B ensures the correctness of WER+ even when inactive SIMD lanes are revived for usage.

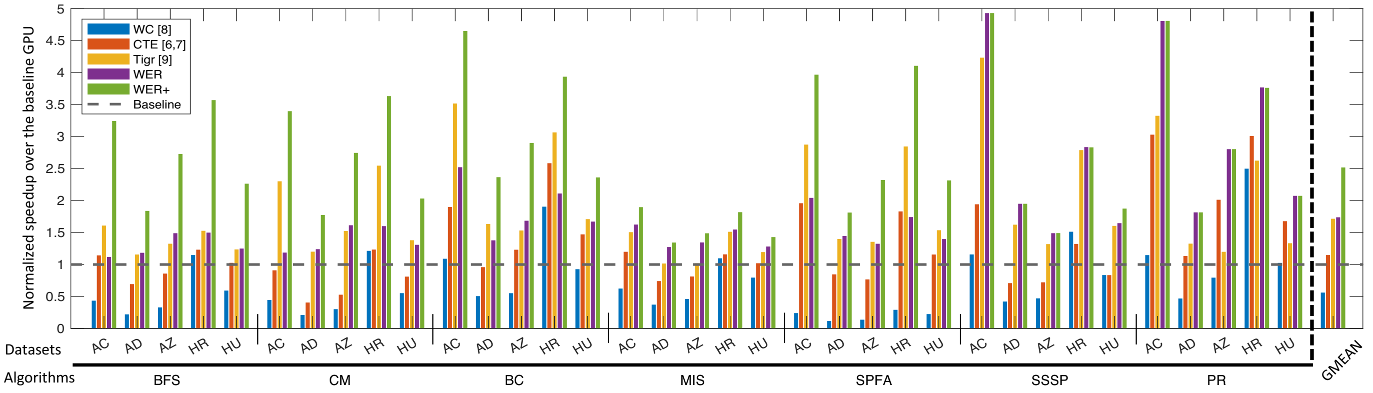


Fig. 7. Normalized performance speedup of various approaches with respect to the baseline GPU for seven representative graph algorithms.

TABLE I
THE CONFIGURATION OF THE BASELINE GPU IN OUR EXPERIMENTS.

# of streaming multiprocessors (SM)	30
# of maximum warps / SM	32
Warp scheduler policy	Greedy-Then-Oldest
# of warp schedulers	4
# of registers / SM	32,768
L1 data cache + shared memory	128 KB
WER table size / SM (i.e. Fig. 4 (a)(b))	720 Bytes

TABLE II
IRREGULAR GRAPH DATASETS ADOPTED IN THE EXPERIMENTS.

	# of vertices	# of edges	Average degree
coAuthorsCiteseer(AC) [16]	0.237M	1.63M	7
coAuthorsDBLP(AD) [16]	0.299M	1.96M	6
amazon0302(AZ) [17, 18]	0.262M	2.47M	9
HR_edges(HR) [17, 19]	0.054M	1.00M	18
HU_edges(HU) [17, 19]	0.048M	0.45M	9

IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results from our evaluation of WER and WER+, specifically analyzing their performance and energy consumption. In addition, we carry out a latency analysis on the design of the *global operand forwarding bus* to validate its practicality.

A. Experimental Setup

We model the baseline GPU and the proposed WER using GPGPU-Sim 4.2 [13], a cycle-accurate GPU simulator. Table I provides the specification of the baseline GPU, which is configured to be similar to that of NVIDIA RTX 2060. This GPU, featuring compute capability 75, represents the most advanced architecture that GPGPU-Sim currently supports. To estimate energy consumption, we calculate the energy consumption of WER using CACTI [14], and the GPU components using AccelWattch [15]. We compare our work against the baseline GPU and two edge-based solutions: virtual warp-centric programming algorithm (WC) [8] and collaborative task engagement (CTE) [6,7]. Moreover, we also compare to the graph transformation approach, Tigr [9] with a degree bound (K_v) of ten, which is the best parameter setup presented in their original paper. Seven graph algorithms, consisting of six from the Pannotia benchmark suite [4] and the Shortest Path Faster Algorithm (SPFA) [5], are selected as the benchmarks. Five of these algorithms, including BFS, Coloring Max (CM), Betweenness Centrality (BC), Max Independent Set (MIS), and SPFA, encountered the inactive thread issue. On the other hand, Single-Source Shortest Path (SSSP) and PageRank (PR) do not suffer from this inefficiency problem. We evaluate these algorithms on a total of five real-world graph datasets from the DIMACS Implementation Challenge [16] and SNAP graph collection [17], with details provided in Table II. These datasets span a variety of domains, including social networks and citation data. The irregular graph structures inherent in these real-world datasets reinforce the validity and broad applicability of our proposed WER and WER+.

B. Performance Analysis and Comparison to the Baselines

Fig. 7 presents the performance speedup of WER and WER+ compared to the baseline GPU and several SOTA solutions. It can be observed that WER, by redistributing the workload among the active SIMD lanes, is able to achieve a geometric mean (abbreviated as 'GMEAN') speedup of 1.74× over the baseline GPU. As for other SOTA methods, our insights from the experiments indicate that WC [8], due to the typically fewer than 32 edges in the average degree of real-world datasets (i.e., Table II), often fails to fully utilize the available SIMD lanes. This limitation hinders its performance, often resulting in it being unable to even match the performance of the baseline GPU. Meanwhile, CTE experiences performance degradation due to the binary searching in each iteration. Although Tigr's data layout addresses intra-warp load imbalance and achieves performance comparable to WER, it does not address the inactive thread issue, resulting in underutilization of computational resources. WER+ further optimizes WER by modifying the SIMT stack to activate all threads to participate in workload processing, thus pushing the performance of WER to achieve a speedup of 2.52×. Particularly for algorithms that suffer from the inactive thread issue (i.e., BFS, CM, BC, MIS, and SPFA), WER+ can attain a speedup of 1.67× over WER. On the whole, WER+ achieves a GMEAN speedup of 2.52× and 1.47× over the baseline GPU and the best SOTA method across seven representative graph algorithms, respectively.

C. Energy Evaluation and Efficiency Validation

Table III presents the average normalized energy consumption for executing graph applications across five dif-

TABLE III
Normalized energy consumption of WER+ and the baseline.

	BFS	CM	BC	MIS	SPFA	SSSP	PR	GMEAN
Baseline GPU	1	1	1	1	1	1	1	1
WER+	0.446	0.439	0.456	0.690	0.433	0.470	0.531	0.495

TABLE IV
Analysis of normalized energy consumption for CM.

	AC	AD	AZ	HU	HR	GMEAN
Baseline	1	1	1	1	1	1
WER+	0.444	0.500	0.423	0.497	0.349	0.439
WER+'s Table	0.00001	0.00001	0.00004	0.00003	0.00001	0.00002

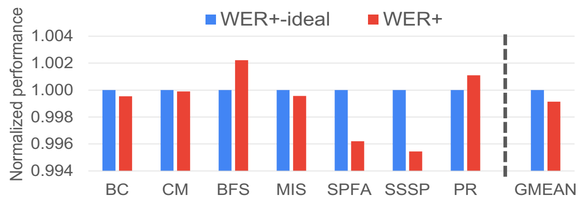


Fig. 8. Normalized performance between WER+-ideal and WER+.

ferent graph datasets. WER+ achieves a geometric mean energy reduction of over 50% compared to the baseline GPU, highlighting its efficiency enhancements. Table IV further offers a comparison of the total normalized energy consumption for WER+ and the tables (i.e., Fig. 4 (a)(b) in Section III.B) for the CM algorithm. The compact 720-byte table size per SM, as shown in Table I, results in negligible energy consumption. These findings validate that the overhead induced by WER+ is minimal, whereas the energy efficiency benefits from WER+ are substantial.

D. Latency Analysis of the Operand Forwarding Bus

Fig. 8 compares the performance of WER+-ideal and WER+, with WER+-ideal representing an ideal version of WER+ that forwards data without any overhead. The results suggest that on average, the difference between them is less than 0.1%. Given that only few global operands require forwarding, the forwarding bus introduces a negligible latency overhead. Moreover, since graph algorithms are often memory-bound, any latency can be effectively overlapped by warp schedulers. As a result, the performance impact of WER+'s forwarding bus is insignificant.

V. Conclusion

In this work, we introduced WER and WER+, two novel solutions crafted to address the challenges in handling irregular graphs on GPGPUs, specifically the inactive threads and intra-warp load imbalance issues. Our approach uniquely combines software and hardware elements in the WER framework to effectively mitigate the issue of intra-warp imbalance. We further enhance computational efficiency with the WER+ extension, which strategically employs inactive threads. The robustness and effectiveness of these innovations were rigorously tested, revealing a significant speedup compared to the baseline GPU and the existing SOTA methods and considerable energy savings.

Acknowledgments

The authors acknowledge the support from the National Science and Technology Council in Taiwan under grant numbers MOST 111-2223-E-007-004-MY3 and 111-2221-E-A49-131-MY3, as well as the support from Google Inc.

References

[1] B. Rozemberczki, C. Allen, and R. Sarkar, "Multi-scale attributed node embedding," *J. Complex Networks*, vol. 9, no. 2, Apr. 2021.

[2] 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, Jan. 2018.

[3] K. Shamsi, Y. R. Gel, M. Kantarcioglu, and C. G. Akcora, "Chartalist: Labeled graph datasets for utxo and account-based blockchains," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2022, pp. 1–14.

[4] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Proc. Int. Symp. on Workload Characterization*, 2013, pp. 185–195.

[5] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. on the Theory of Switching*, 1959, pp. 285–292.

[6] F. Khorasani, B. Rowe, R. Gupta, and L. N. Bhuyan, "Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement," in *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, May 2016, pp. 524–533.

[7] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 39–50.

[8] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 267–276, Aug. 2011.

[9] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, Mar. 2018, pp. 622–636.

[10] Y. Lü, H. Guo *et al.*, "GraphPEG: Accelerating graph processing on GPUs," *ACM Trans. on Architecture and Code Optimization*, vol. 18, no. 3, pp. 1–24, May 2021.

[11] A. Segura, J.-M. Arnau, and A. González, "SCU: A gpu stream compaction unit for graph processing," in *Proc. Int. Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 424–435.

[12] NIRVANA, "Maxas SASS Assembler," [Online]. Available: https://github.com/NervanaSystems/maxas, 2016.

[13] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. Int. Symp. on Computer Architecture (ISCA)*, May-Jun. 2020, pp. 473–486.

[14] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. Int. Symp. on Microarchitecture (MICRO)*, Dec. 2007, pp. 3–14.

[15] V. Kandiah, S. Peverelle *et al.*, "AccelWattch: A power modeling framework for modern GPUs," in *Proc. Int. Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 738–753.

[16] DIMACS, "DIMACS10 graph partitioning challenge," [Online]. Available: https://sparse.tamu.edu/DIMACS10, 2010.

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[18] J. Leskovec, L. A. Adamic, and B. Adamic, "The dynamics of viral marketing," *ACM Trans. on the Web*, vol. 1, no. 1, pp. 1–39, May 2007.

[19] B. Rozemberczki, R. Davies, R. Sarkar, and C. Sutton, "GEMSEC: Graph embedding with self clustering," in *Proc. IEEE/ACM Int. Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*, Aug. 2019, pp. 65–72.