WER: Maximizing Parallelism of Irregular Graph Applications Through GPU **W**arp **E**qualize**R**

Session number: 3A-2

En-Ming Huang[1], Bo-Wun Cheng[1], Meng-Hsien Lin[2], Chun-Yi Lee[1], and Tsung Tai Yeh[2]

[1]Elsa lab, National Tsing Hua University, Hsinchu, Taiwan
[2]CAS Lab, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE

Contributions

- Problem:
  Mapping irregular graph algorithms to single-instruction multiple-threads (SIMT) programming model (e.g., NVIDIA GPUs)
- We address the “inactive threads” and “intra-warp load imbalance” issues incurred by irregular graph algorithms
- We propose a software-hardware co-optimization solution, with lightweight hardware modification to the GPU pipeline
- Our proposed approach enhances the overall performance of graph algorithms
  - Compare to baseline implementation: 2.5x
  - Compare to state-of-the-art methodologies: 1.5x

Outline

1. Introduction
2. Methodology:
    - Warp EqualizeR (WER): Addressing intra-warp workload imbalance issue
    - WER+: Addressing inactive thread issue
3. Experimental Results
4. Conclusion

Graphics Processing Unit (GPU)

- Parallel processors that are originally built for graphics rendering
- GPUs are increasingly utilized in general-purpose computing due to the following reason:
  - Parallel processing power
  - Ease of programming
- Workloads that take advantage of GPUs' computing power:
  - Machine learning
  - Graph algorithm
  - Scientific computing

Single-Instruction Multiple-Thread (SIMT) Model

- Typical GPUs are programmed using the SIMT programming model
- SIMT is similar to single-instruction multiple-data (SIMD), but allows **branching condition**
- The GPU program is written from the view of “thread”
- Properties of NVIDIA GPUs:
  - SIMD width is 32
  - 32 threads are grouped into a warp
  - Warp size is 32
- Threads within the same warp execute the same instruction

SIMT: Branch Divergence Problem

- A warp executes instructions in a **lock-step** fashion, allowing branching instructions and divergence behavior
- Threads that do not satisfy the branching condition are disabled and become idled
- Branch divergence problem also happens on **loops** with different iterations

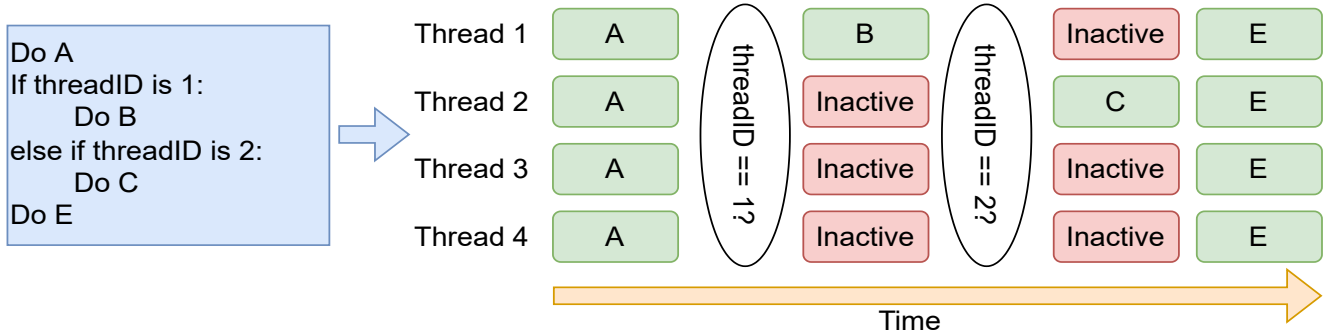


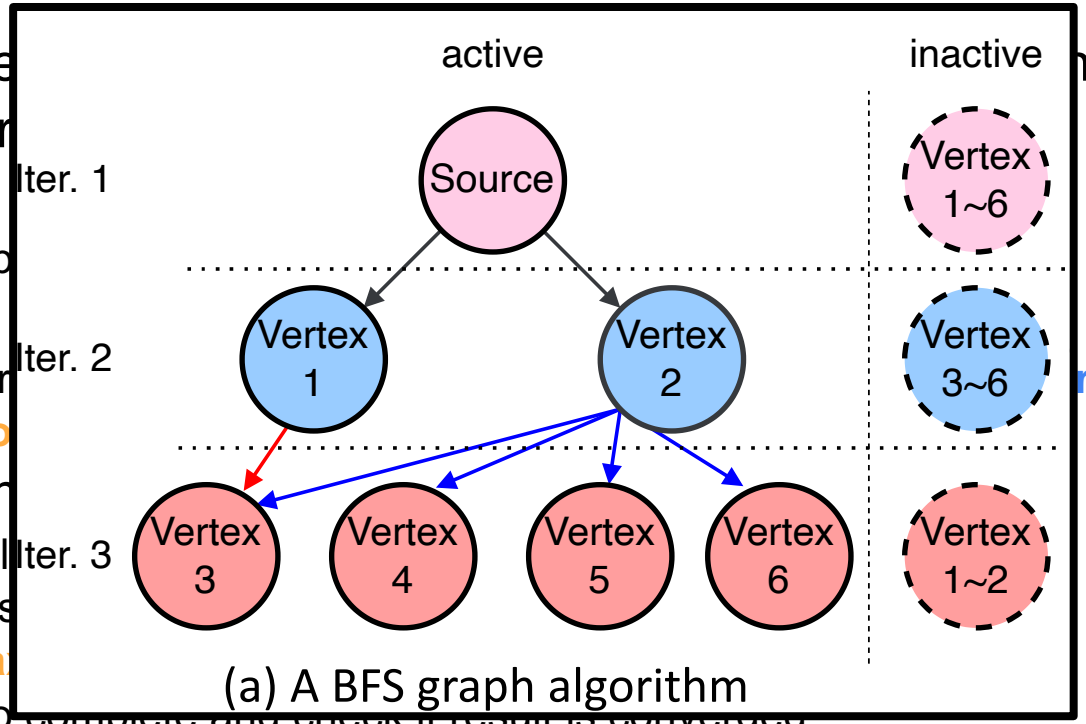
Fig. 1. An illustration of the branch divergence within a 4-threaded warp

Motivation

- Graph algorithms often requires performing “**relaxing**” operation on **some vertices**, and repeating until the result is converged.
- **Relaxing operation**:
  - Iterate over all neighbors, and update their values according to the algorithm
- **Activated vertices**:
  - Depending on the current state, the algorithm should select **some specific vertices** to perform **relaxing operation** on their neighbors.
- Common method to map graph algorithms to GPU:
  1. Each GPU thread selects a vertex
  2. Each thread identifies whether its vertex is **activated**.
  3. If ”Yes”, perform **relaxing operation** on its neighbor.
  4. Wait for all threads to complete and check if result is converged.
     If not, go to (1).

Motivation

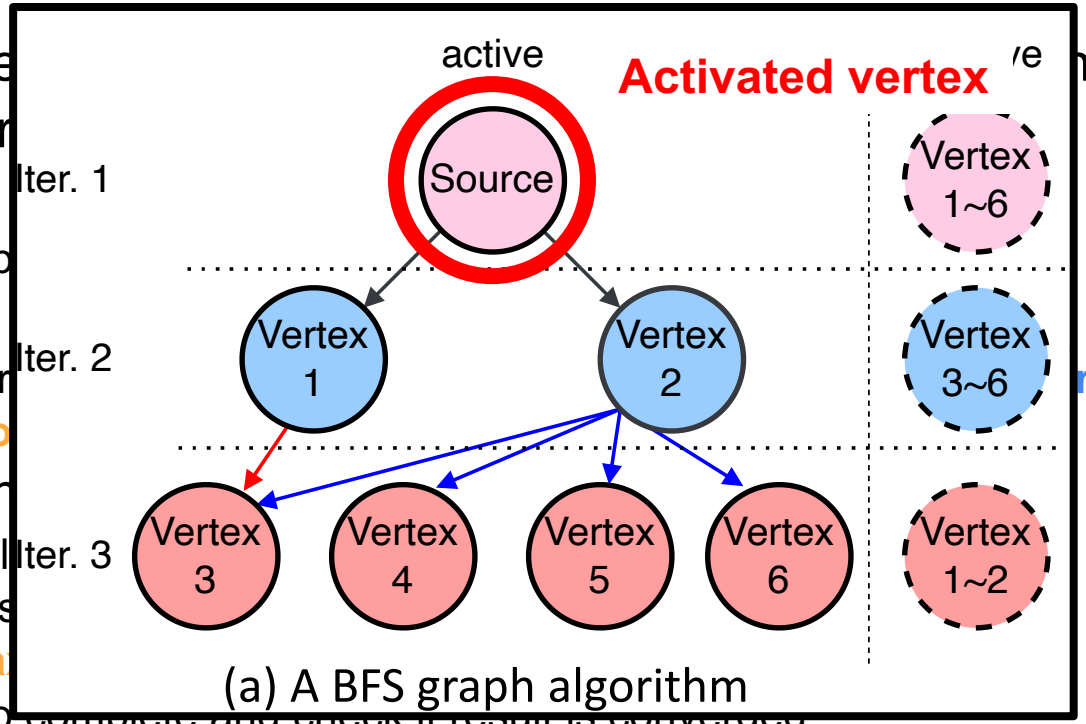
- Graph algorithms ofte
  some vertices, and r
- **Relaxing operation**:
  - Iterate over all neighb
- **Activated vertices**:
  - Depending on the cur
    to perform **relaxing o**
- Common method to r
  1. Each GPU thread sel
  2. Each thread identifies
  3. If ”Yes”, perform **rela**
  4. Wait for all threads to complete and check if result is converged.
     If not, go to (1).



(a) A BFS graph algorithm

Motivation

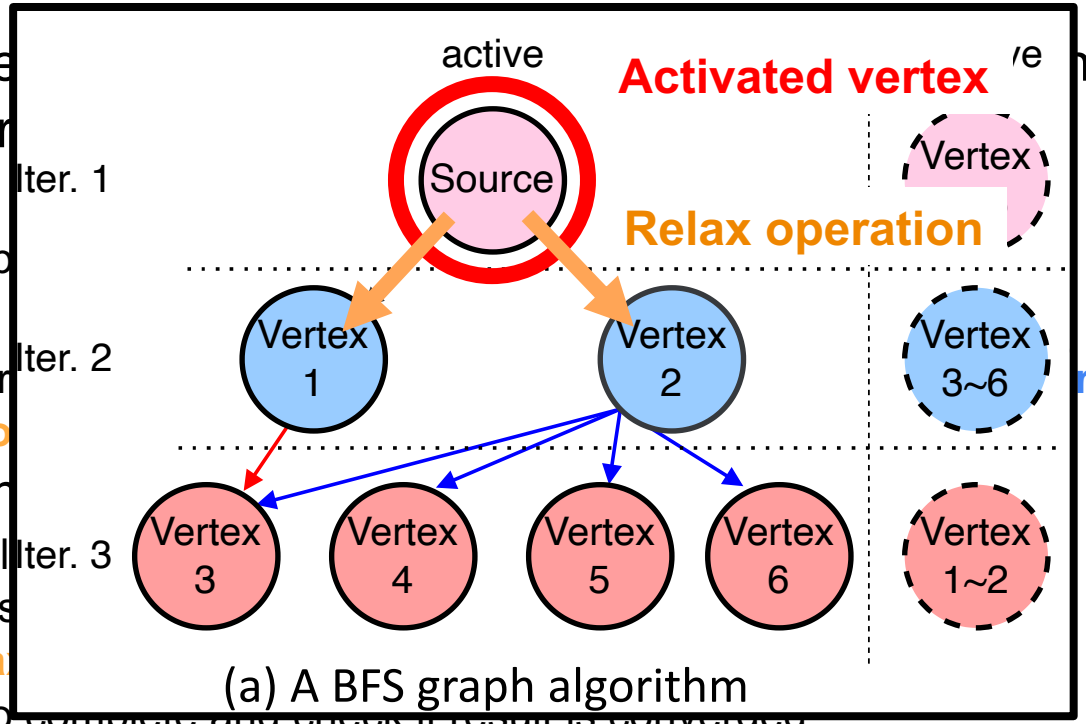
- Graph algorithms ofte … some vertices, and r …
- **Relaxing operation**:
  - Iterate over all neighb …
- **Activated vertices**:
  - Depending on the cur … to perform relaxing o …
- Common method to r …
  1. Each GPU thread sel …
  2. Each thread identifies …
  3. If "Yes", perform rela …
  4. Wait for all threads to complete and check if result is converged. If not, go to (1).



(a) A BFS graph algorithm

Motivation

- Graph algorithms ofte … some vertices, and r …
- **Relaxing operation**:
  - Iterate over all neighb …
- **Activated vertices**:
  - Depending on the cur … to perform relaxing o …
- Common method to r …
  1. Each GPU thread sel …
  2. Each thread identifies …
  3. If "Yes", perform rela …
  4. Wait for all threads to complete and check if result is converged. If not, go to (1).



(a) A BFS graph algorithm

Motivation

- Fig. 2(a): an example execution order of **Breadth-First Search** (BFS)
- In each iteration, vertices are classified into two types, **activated** and **inactive**
- Only activated vertices relaxes their neighbors (e.g., marking the vertex as ‘traveled’ in BFS).
- The # of neighbors of each vertex varied
- Fig. 2(b): Approximately 70% or more threads are inactive

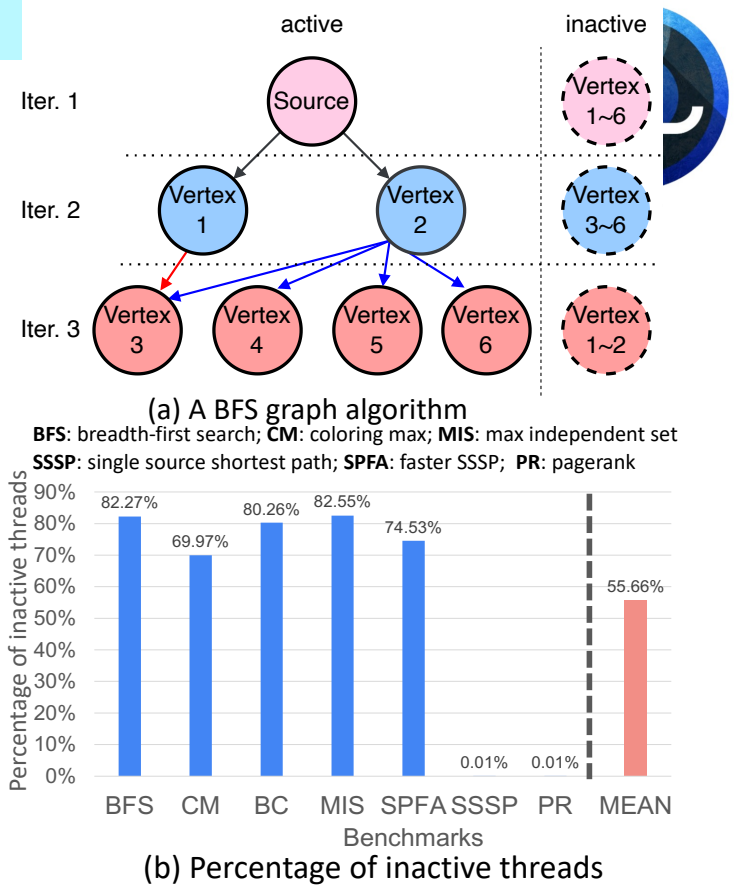


Fig. 2. Challenges in processing irregular graphs on GPUs:
(a) A breadth first search (BFS) graph algorithm for highlighting the inactive threads and intra-warp load imbalance issues
(b) A percentage breakdown of inactive threads in seven graph algorithms.

Key Challenges and the Proposed Solution

- **Intra-warp load imbalance**:
  the # of neighbors for each vertex varied, leading to idled SIMD lane
- **Inactive thread**: some threads are deactivated due to the graph algorithm and the given graph

- **We propose a hardware and software synergistic approach.**
- **Our proposed approach enhances the overall performance of graph algorithms**
  - **Compare to baseline implementation: 2.5x**
  - **Compare to state-of-the-art methodologies: 1.5x**

Warp EqualizeR (WER):
Addressing intra-warp workload imbalance issue

Warp EqualizeR (WER) Overview

- We propose a synergistic software and hardware framework to tackle the challenges of **intra-warp load imbalance**
- Common implementations map a vertex to a thread (vertex-based scheme), leading to **intra-warp load imbalance**
- Two **software intrinsics** (push job & pop job) are proposed for redistributing workloads by our WER **hardware**

	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	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

(b) WER.

Fig. 3. An illustrative comparison of task distribution schemes

```
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

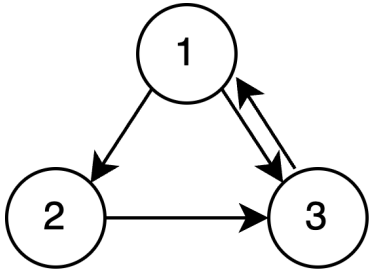
```
1.__global__ void color1(....) {
2.  int tid = blockIdx.x * blockDim.x + threadIdx.x;
3.  __shared__ int maximum[256];
4.  if (tid < num_nodes){
5.    if (color_array[tid] == -1) {
6.      int start = row[tid]; int end;
7.      ...
8.      push_job (start, end); // WER intrinsic
9.      while(pop_job(&edge)) { // 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. 4. A GPU kernel function vs. the one with WER intrinsics

Software Design

- Graph can be expressed by adjacent lists, which store the neighbors for each vertex



0	1	2	3
2	3	3	1

Vertex 1:	2	3
Vertex 2:	3	
Vertex 3:	1	

Concatenate

	Start index	End index
Vertex 1	0	2
Vertex 2	2	3
Vertex 3	3	4

Software Design

- Our WER architecture required a table for storing the characteristics of the workloads
- Push_job:
  The # of workloads and workloads ID are recorded by push_job instruction.
  In particular, the “**start index**” and “**end index**” is recorded by push_job.
- Pop_job:
  Each thread extract a single job from the WER architecture

	Start index	End index
Vertex 1	0	2
Vertex 2	2	3
Vertex 3	3	4

Hardware Design

- The colored blocks are the modified components in the GPU pipeline.

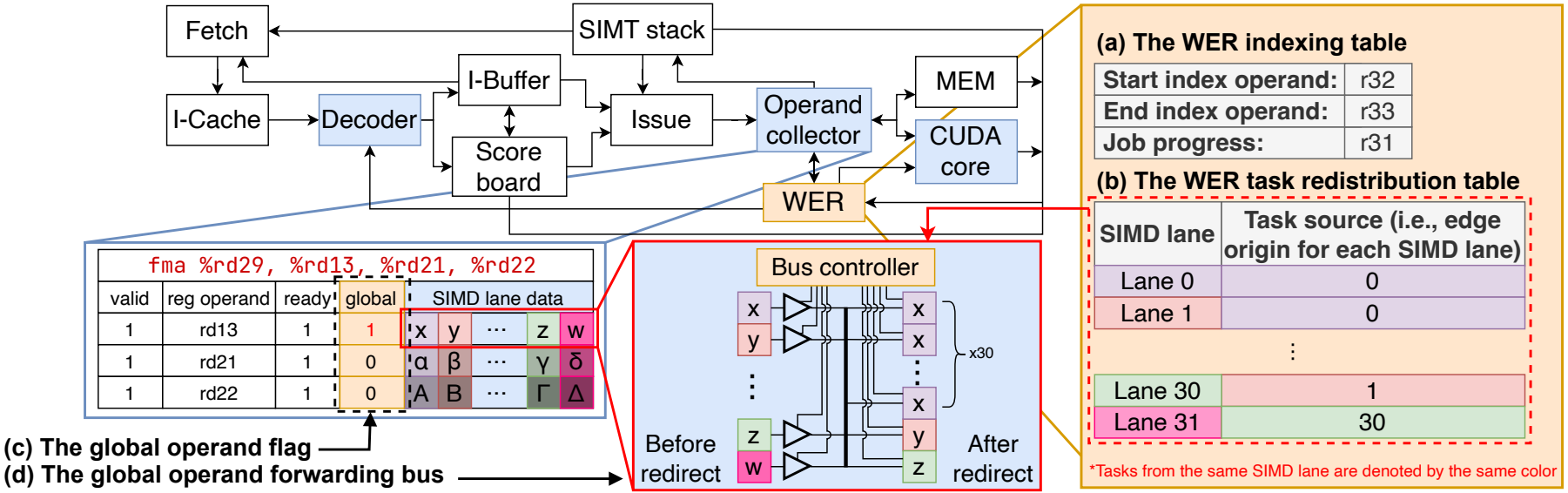


Fig. 5. The WER-enhanced GPU architecture. In this example, the operands `r32` and `r33` store the index bounds in the register file. Threads 0~29 are processing the edges from lane 0, while the edges in lanes 1 and 30 are redistributed to threads 30 and 31, respectively.

Hardware Design

- (a): The table used for cooperating with the WER instructions

```
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]; }}
```

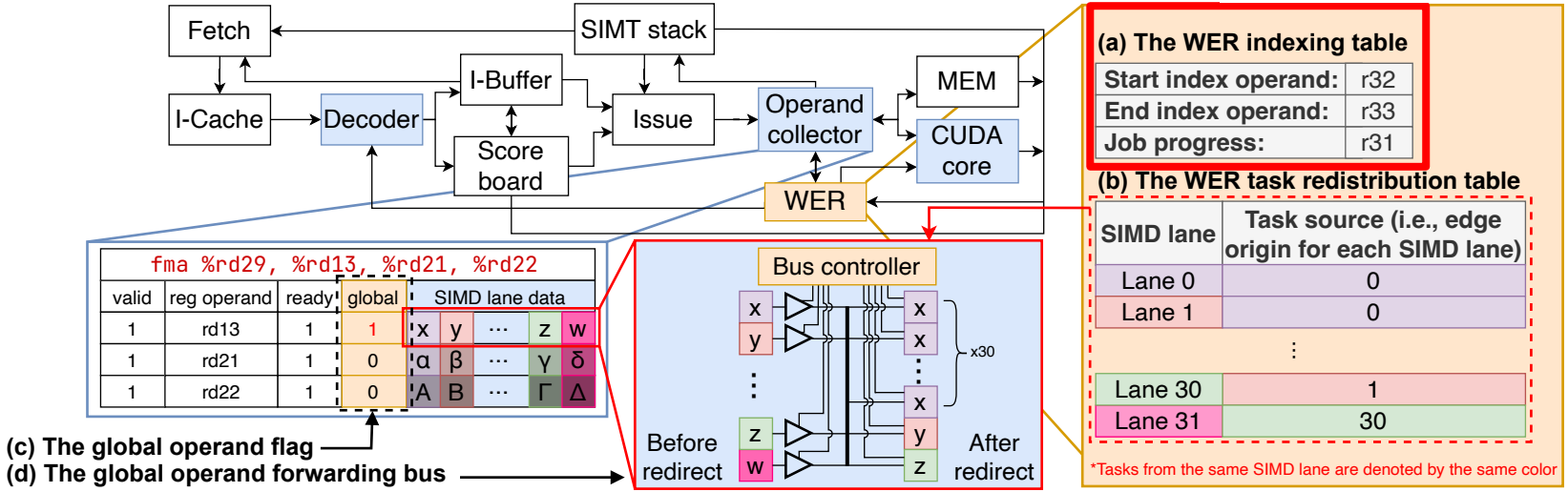


Fig. 5. The WER-enhanced GPU architecture. In this example, the operands r32 and r33 store the index bounds in the register file. Threads 0~29 are processing the edges from lane 0, while the edges in lanes 1 and 30 are redistributed to threads 30 and 31, respectively.

Hardware Design

- (b)~(d): Designed for addressing data dependency issues between threads

	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	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. 3. An illustrative comparison of task distribution schemes.

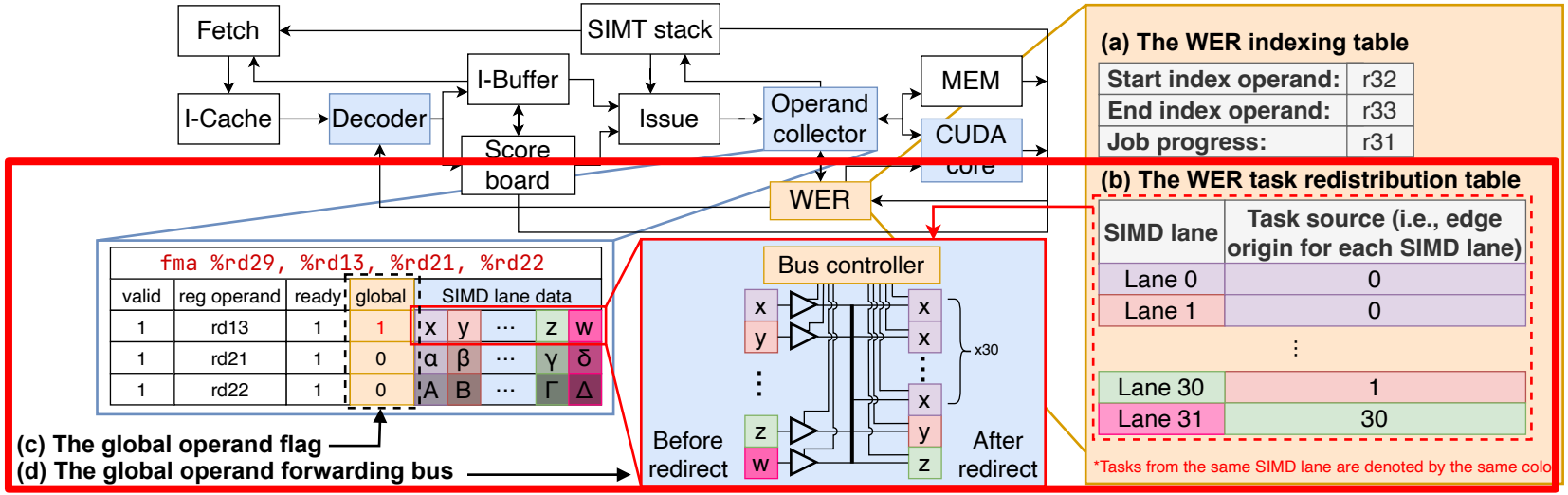


Fig. 5. The WER-enhanced GPU architecture. In this example, the operands r32 and r33 store the index bounds in the register file. Threads 0~29 are processing the edges from lane 0, while the edges in lanes 1 and 30 are redistributed to threads 30 and 31, respectively.

Hardware Design: Addressing data dependency

- Lane 0~2 execute the jobs from the same vertex (lane 0)
- Some of the data is stored in lane 0's register
- In this example, **threadIdx.x** is a register that stores the index of each SIMD lane

	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	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

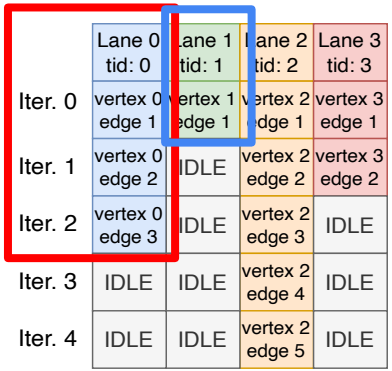
(b) WER

For lane 0: threadIdx.x = 0
For lane 1: threadIdx.x = 1

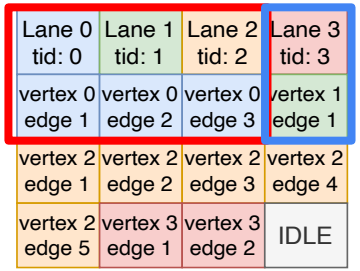
```
1.__global__ void color1(....) {
2.  int tid = blockIdx.x * blockDim.x + threadIdx.x;
3.  __shared__ int maximum[256];
4.  if (tid < num_nodes){
5.    if (color_array[tid] == -1) {
6.      int start = row[tid]; int end
7.    ...
8.    push_job (start, end); // WER intrinsic
9.    while(pop_job(&edge)) { // 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]; }}
```

Hardware Design: Addressing data dependency

1. Identify the source lane (e.g., 0 for lanes 0~2; 1 for lane 3)
2. Identify which register has data dependency issue
   (if not, then we do not care)
3. Design a mechanism to forward the data



(a) vertex-based scheme



(b) WER

For lane 0: threadIdx.x = 0
For lane 1: threadIdx.x = 1

```
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]; }}
```

Hardware Design: Addressing data dependency

1. We first identify the source lane by Fig. 5. (b)
2. Next, identify which register has data dependency issue: Fig. 5. (c)
   (if not, then we do not care)
3. We propose a forwarding bus to forward the data in Fig. 5. (d)

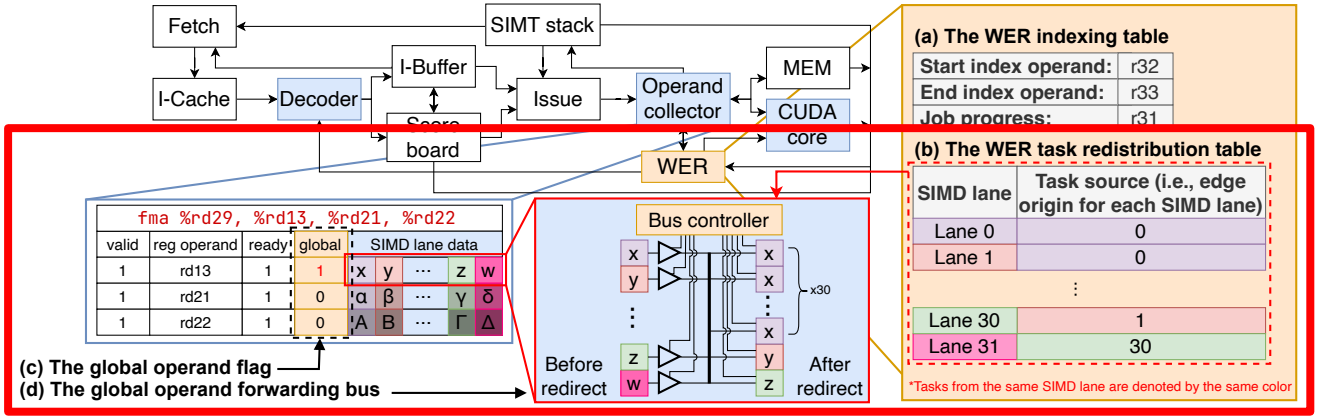


Fig. 5. The WER-enhanced GPU architecture. In this example, the operands `r32` and `r33` store the index bounds in the register file. Threads 0~29 are processing the edges from lane 0, while the edges in lanes 1 and 30 are redistributed to threads 30 and 31, respectively.

WER+:
Addressing inactive thread issue

Inactive Threads

- WER only redistributed workloads to the activated threads
- Threads that were disabled outside the for loop do not participate in the computation

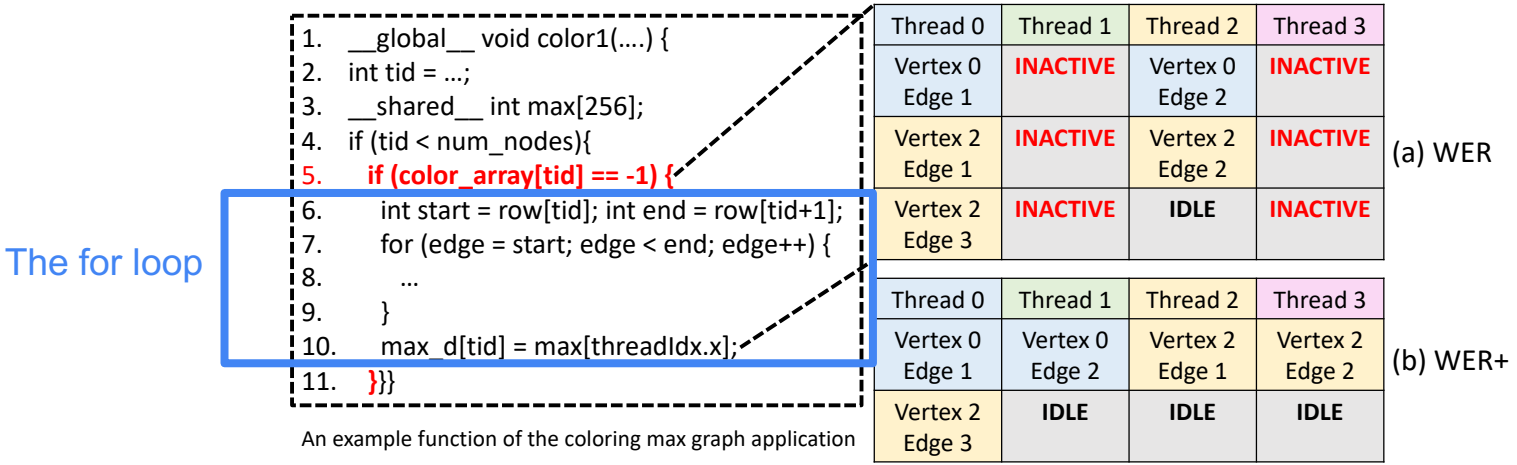


Fig. 6. The inactivate threads problem in WER

WER+: Modified SIMT stack

- SIMT stack: the hardware that manages the which SIMD lane is active.
  - Used to masked out SIMD lanes that does not satisfy the branch condition
- In WER+, we insert a mask with all bit set to the SIMT stack, notifying the hardware that all SIMD lanes should participated the compuation.

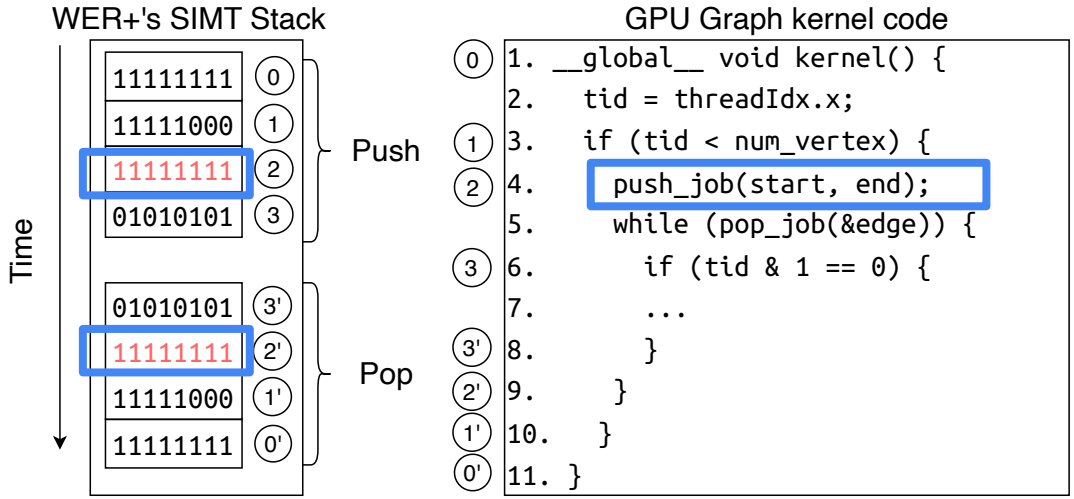


Fig. 7. An illustration of the SIMT stack

Experimental Results

Experimental Setup

- We perform the simulation by GPGPU-SIM[1, 2]
- GPGPU-SIM: a cycle-level simulator that models contemporary graphics processing units
- We evaluate 7 graph algorithms on 5 real-world datasets
- Algorithms:
  - BFS, Coloring Max (CM), Betweenness Centrality (BC), Max Independent Set (MIS), Shortest Path Faster Algorithm (SPFA), Bellman-Ford (SSSP), PageRank (PR)
- Datasets:

	Number of vertices	Number of edges
coAuthorsCiteseer (AC)	0.24M	1.63M
coAuthorsDBLP (AD)	0.30M	1.96M
amazon0302 (AZ)	0.26M	2.47M
HR_edges (HR)	0.05M	1.00M
HU_edges (HU)	0.05M	0.45M

[1]A. Bakhoda, *et al.*, ***Analyzing CUDA workloads using a detailed GPU simulator,*** ISPASS, 2009
[2]M. Khairy, *et al.*, ***Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling,*** ISCA, 2020

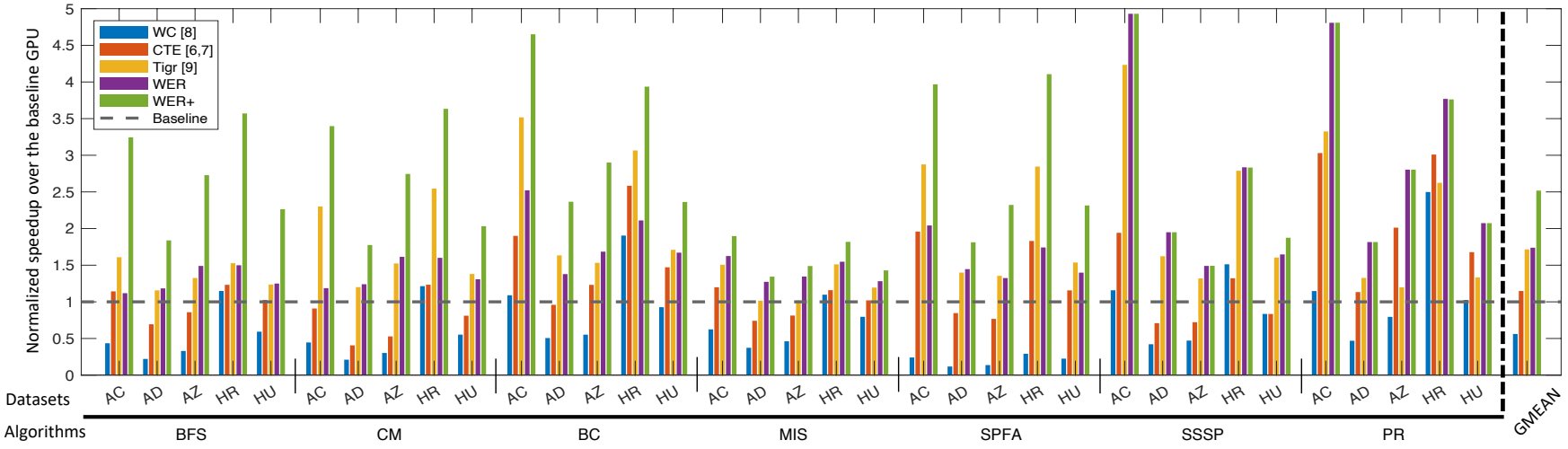
Experimental Setup

- Baseline GPU implementation:
  - The benchmarks used in the GPGPU-Sim paper[1]
- State-of-the-art (SOTA) methodologies:
  - Virtual warp-centric programming algorithm (WC)[3]
  - Collaborative task engagement (CTE) [4]
  - Transforming Irregular Graphs for GPU-Friendly Graph Processing (Tigr) [5]

[3]S. Hong, *et al.*, ***Accelerating CUDA graph algorithms at maximum warp***. SIGPLAN, 2011
[4]F. Khorasani, *et al.*, **Eliminating Intra-Warp Load Imbalance in Irregular Nested Patterns via Collaborative Task Engagement**, IPDPS, 2016
[5]A. Sabet, *et al.*, **Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing**, ASPLOS, 2018

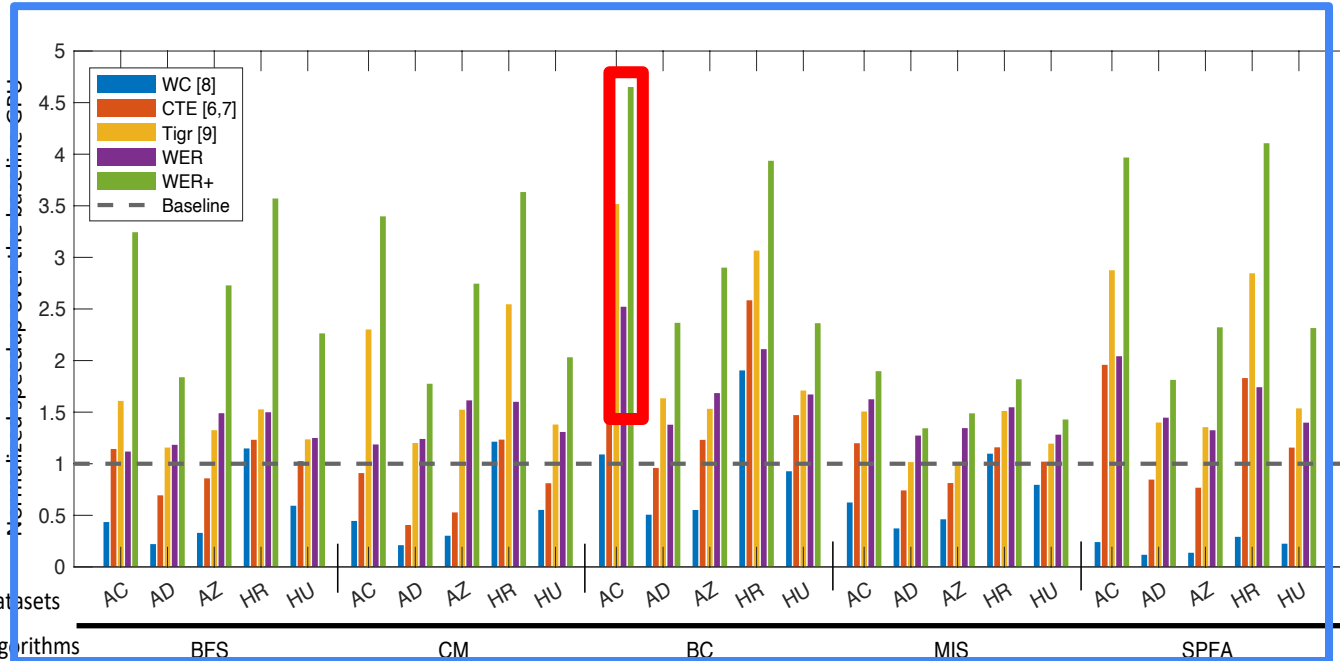
Performance Speedup

- We achieved a geometric mean speedup of **2.52x** and **1.47x** over the baseline GPU and existing state-of-the-art methodologies



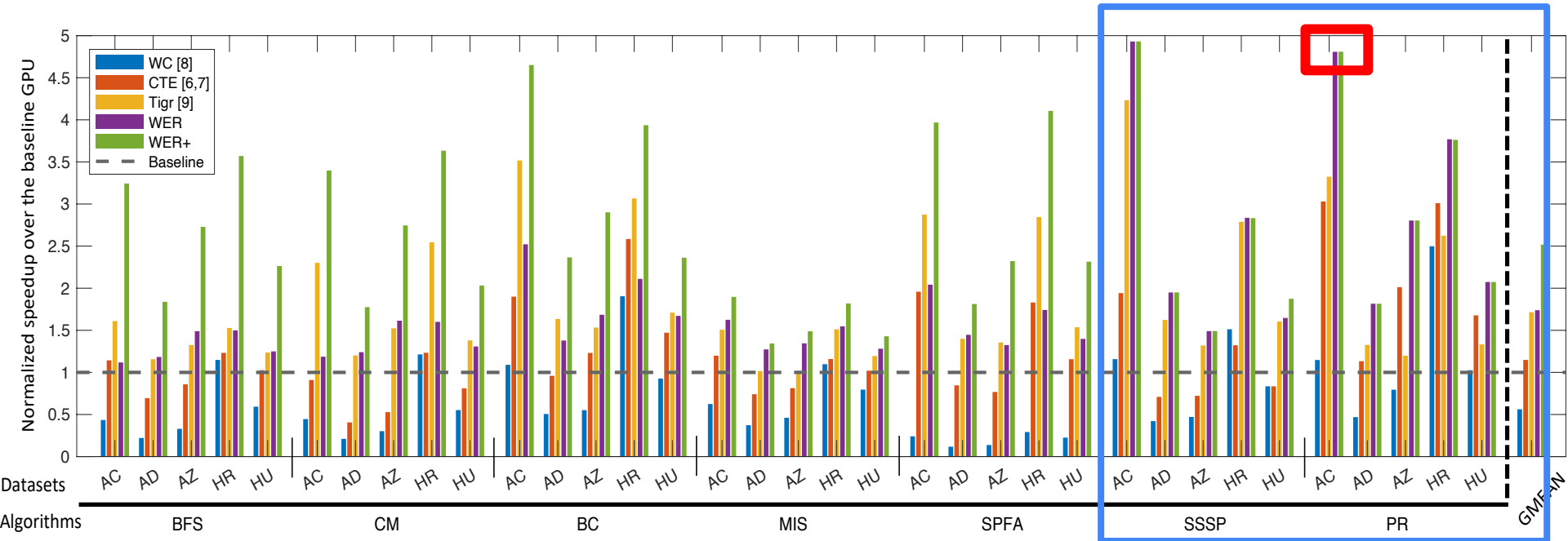
Performance Speedup

- For algorithms that have inactivate thread issue, **WER+** output performs **WER+** by **1.7x** speedup.



Performance speedup

- For algorithms that **do not** have inactivate thread issue, **WER+** and **WER+** have similar performance.



Energy Consumption

- We achieved a geometric mean speedup of **2.5x**
- The program runtime is reduced by 2.5x. Therefore, decreasing the energy consumption to **50%**

Normalized energy consumption
(including the introduced hardware circuits and tables)

	BFS	CM	BC	MIS	SPFA	PR	GEOMETRIC MEAN
Baseline	1	1	1	1	1	1	1
WER+	0.45	0.44	0.46	0.69	0.43	0.47	0.50

Forwarding bus latency

WER+-ideal: the data could be forwarded in a cycle
WER+: Requires **N** cycles for forwarding data from **N** different sources

We show that the forwarding bus does not incur additional overhead.

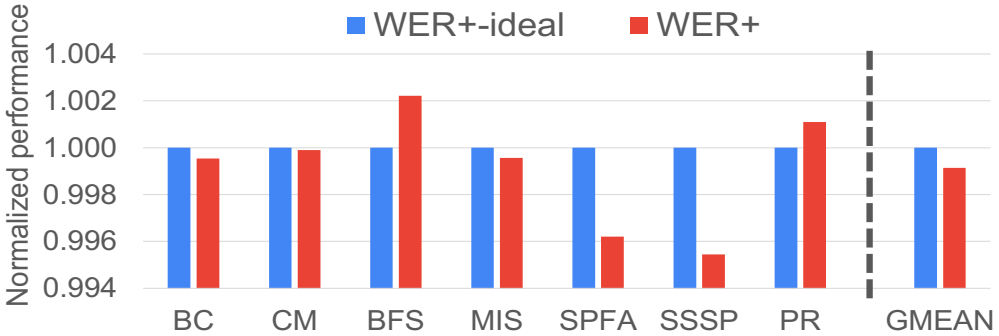


Fig. 8. The performance comparison between **WER+**-ideal and **WER+**

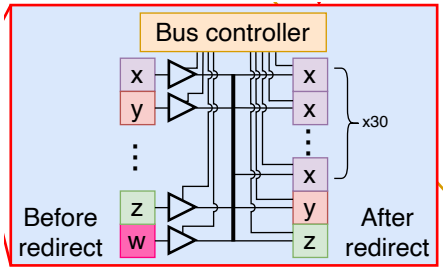


Fig. 9. The introduced forwarding bus

Limitations and Future Work

- Currently, only graph algorithms could be applied
- Future work: support dynamic scheduling for applications that the number of workloads is unknown. For instance, ray tracing.

References

1. A. Bakhoda, *et al.*, ***Analyzing CUDA workloads using a detailed GPU simulator***, ISPASS, 2009
2. M. Khairy, *et al.*, ***Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling***, ISCA, 2020
3. S. Hong, *et al.*, ***Accelerating CUDA graph algorithms at maximum warp***. SIGPLAN, 2011
4. F. Khorasani, *et al.*, ***Eliminating Intra-Warp Load Imbalance in Irregular Nested Patterns via Collaborative Task Engagement***, IPDPS, 2016
5. A. Sabet, *et al.*, ***Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing***, ASPLOS, 2018

WER: Maximizing Parallelism of Irregular Graph Applications Through GPU **W**arp **E**qualize**R**

Session number: 3A-2

Thank you for listening

En-Ming Huang[1], Bo-Wun Cheng[1], Meng-Hsien Lin[2], Chun-Yi Lee[1], and Tsung Tai Yeh[2]

[1]Elsa lab, National Tsing Hua University, Hsinchu, Taiwan
[2]CAS Lab, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE

Questions: emhuang@m109.nthu.edu.tw