# Efficient Maximal Biclique Enumeration on GPUs

Zhe Pan, Shuibing He, Xu Li, Xuechen Zhang[*], Rui Wang, Gang Chen

Zhejiang University, [*]Washington State University Vancouver

# Outline

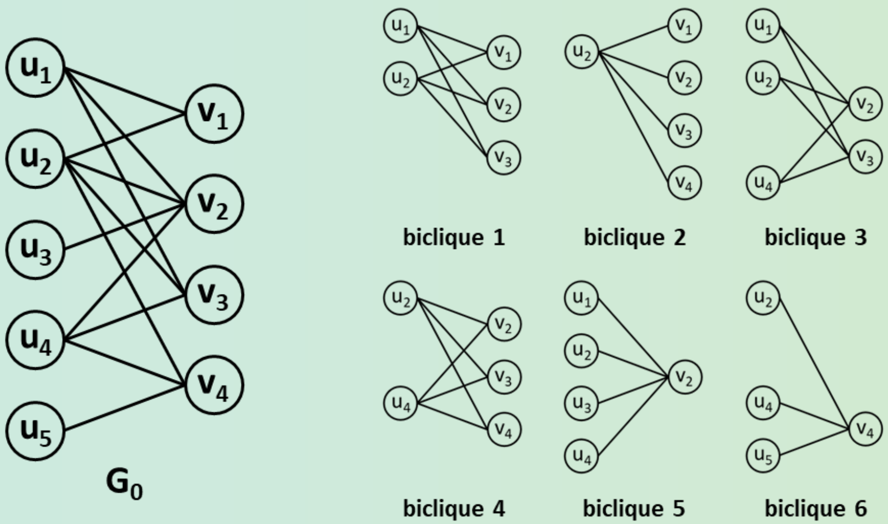
- **Introduction**
  - Problem definition
  - MBE on CPUs
  - Related work comparison
- **Challenges of MBE on GPUs**
  - Large memory requirement
  - Massive thread divergence
  - Load imbalance
- **GMBE : the first highly-efficient GPU solution for the MBE problem**
  - Stack-based iteration with node reuse
  - Pruning using local neighborhood sizes
  - Load-aware task scheduling
- **Evaluation**

# Introduction

- Problem definition
- MBE on CPUs
- Related work comparison

# Introduction : Problem Definition

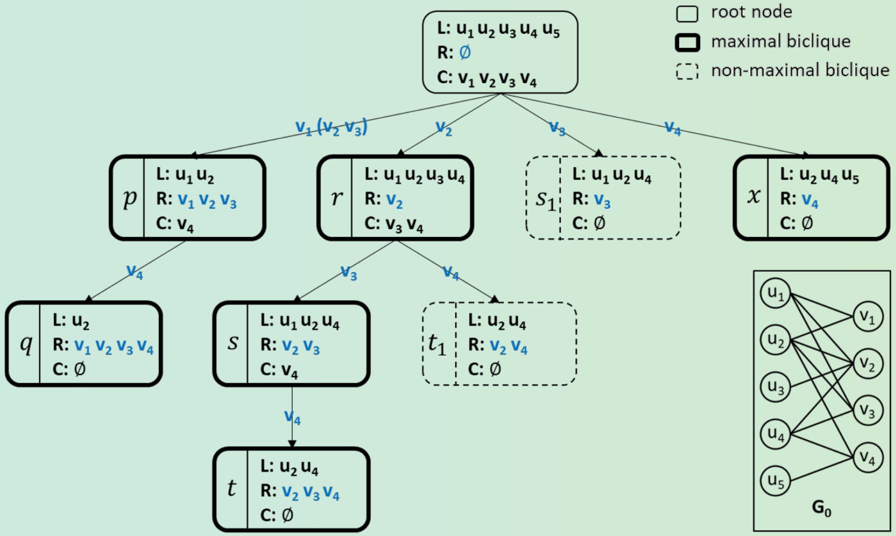
- **Preliminaries**
  - Bipartite graph $G(U, V, E)$ : A graph structure contains two disjoint vertex sets $U, V$ and an edge set $E$. $E \subseteq U \times V$.
  - Biclique : A complete bipartite graph in which every vertex is connected to every vertex in the opposite subset.
  - Maximal biclique : a biclique that can not be further enlarged to form a large biclique.
- **Problem definition**
  - Maximal biclique enumeration (MBE) aims to find all maximal bicliques in $G$.




A bipartite graph $G_0$ containing 6 maximal bicliques.

# Introduction : MBE on CPUs

- **Set enumeration tree for MBE**
  - Each tree node is a 3-tuple $(L, R, C)$. $(L, R)$ is the corresponding biclique and $C$ stores candidate vertices for expanding $R$.
- **Baseline solution**
  - Step 1 : Utilize a set enumeration tree to generate the powerset of $V$.
  - Step 2 : Expand each subset of the powerset of $V$ to a biclique $(L, R)$ and enumerate maximal ones.



# Introduction : MBE on CPUs

- **Recent optimizations**
  - Vertex ordering [1, 2, 5]
  - Candidates pruning using pivots [1, 2]
  - Parallelization on multicore CPUs [3] or distributed architectures [4]

**Existing solutions for MBE are insufficient because their performance speedup is constrained by the limited parallelism of CPUs.**

[1] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, and Jianxin Li. 2022. Efficient Maximal Biclique Enumeration for Large Sparse Bipartite Graphs. VLDB 2022. 1559-1571.

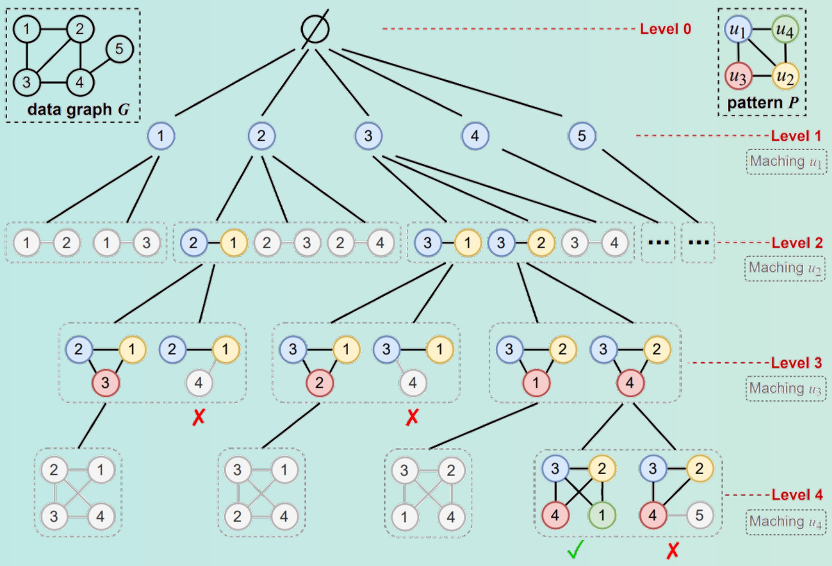
[2] Aman Abidi, Rui Zhou, Lu Chen, and Chengfei Liu. Pivot-Based Maximal Biclique Enumeration. IJCAI 2020. 3558–3564.

[3] Apurba Das and Srikanta Tirthapura. 2019. Shared-Memory Parallel Maximal Biclique Enumeration. HiPC 2019.

[4] Arko Provo Mukherjee and Srikanta Tirthapura. Enumerating Maximal Bicliques from a Large Graph Using MapReduce. IEEE Trans. Serv. Comput. 10, 5 (2017), 771–784.

[5] Yun Zhang, Charles A. Phillips, Gary L. Rogers, Erich J. Baker, Elissa J. Chesler, and Michael A. Langston. BMC bioinformatics 15, 1 (2014), 110.

# Introduction : Related Work Comparison




An enumeration tree for mining pattern P in data graph G.

| Problem | MBE | Graph pattern mining [1] |
|---|---|---|
| Vertex count in enumerated subgraphs | Unfixed number of vertices, can be large. | Fixed number of vertices equivalent to pattern size \|P\|, typically small. |
| Enumeration tree height | Unfixed and can be up to $d_{max}(V)$. | Fixed and equal to \|P\|. |
| Conclusion | (1) MBE requires **significantly more memory** than GPM to actively maintain up to $d_{max}(V)$ tree nodes for backtracking. (2) MBE generates **more severe imbalanced workloads** than GPM due to the variation in height among its enumeration trees. | |

[1] Xuhao Chen and Arvind. Efficient and Scalable Graph Pattern Mining on GPUs. OSDI 2022. 857–877.

# Challenges

- Large memory requirement
- Massive thread divergence
- Load imbalance

# Challenge 1 : Large Memory Requirement

Dynamic memory allocations on GPUs are expensive

↓

Pre-allocate memory on GPUs before enumeration

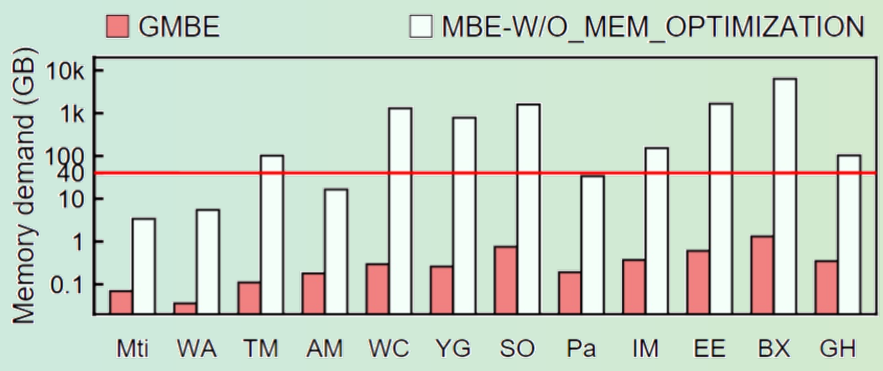
We need to execute multiple MBE procedures in parallel

↓

Requires multiple blocks of memory

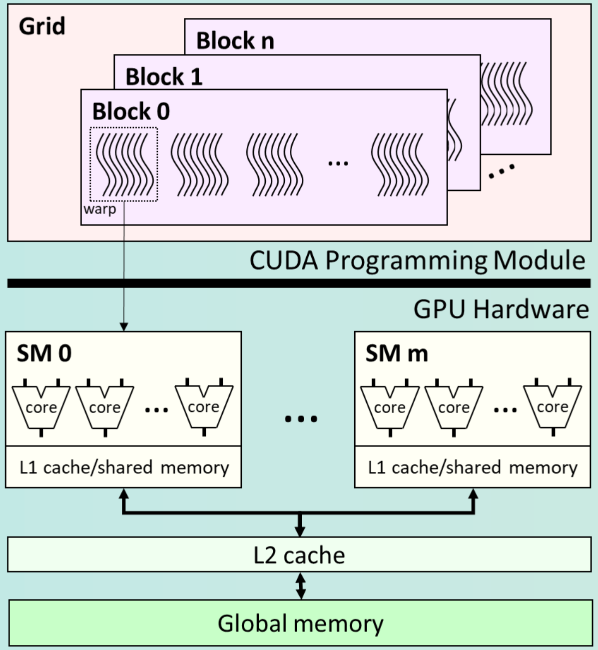
↓

Exceeds the memory capacity




Directly parallelizing existing MBE procedures on an A100 GPU will exceed the memory capacity on multiple datasets.

# Challenge 2 : Massive Thread Divergence




GPU architecture.

**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```

**CS 1**

```
code A;
code A;
```

CS 0 and CS 1 are GPU code segments where threads with different routines execute 2 sets of codes each.

# Challenge 2 : Massive Thread Divergence

**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```

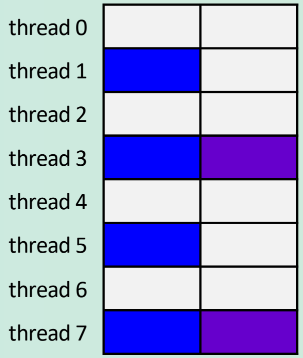

**CS 1**

```
code A;
code A;
```

# Challenge 2 : Massive Thread Divergence

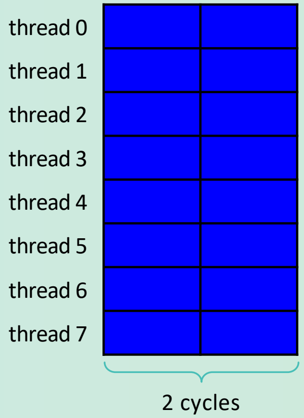
**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```




**CS 1**

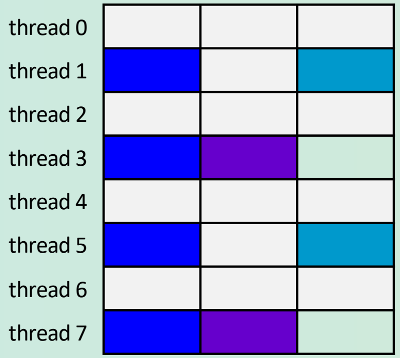
```
code A;
code A;
```



# Challenge 2 : Massive Thread Divergence

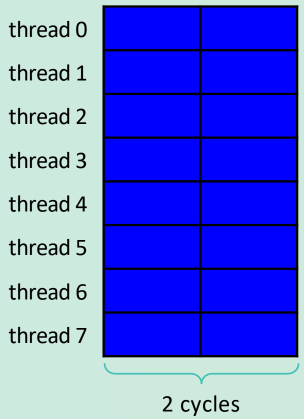
**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```




**CS 1**

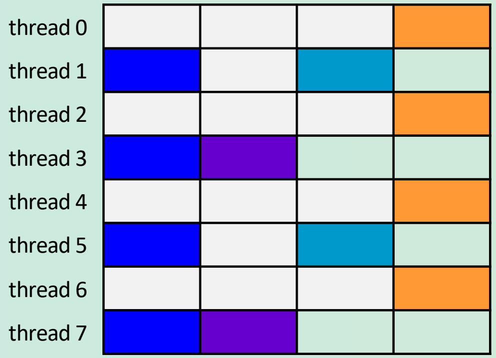
```
code A;
code A;
```



# Challenge 2 : Massive Thread Divergence

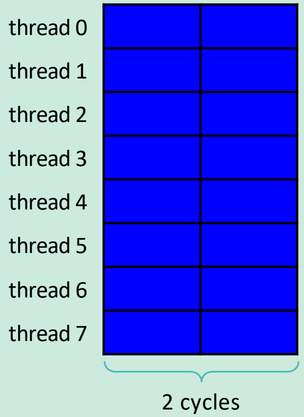
**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```




**CS 1**

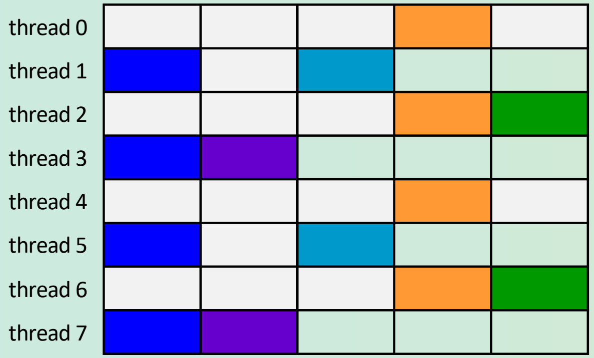
```
code A;
code A;
```



# Challenge 2 : Massive Thread Divergence

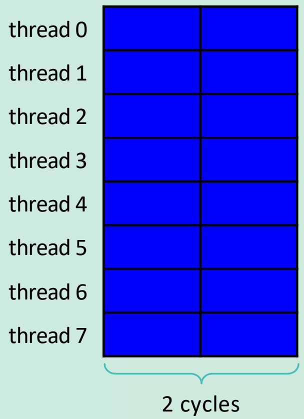
**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```




**CS 1**

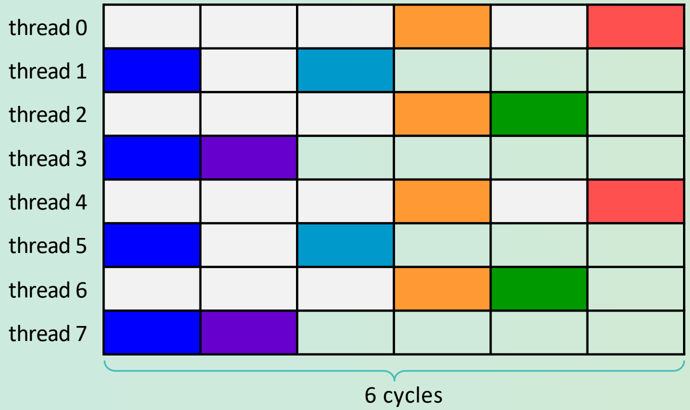
```
code A;
code A;
```



# Challenge 2 : Massive Thread Divergence

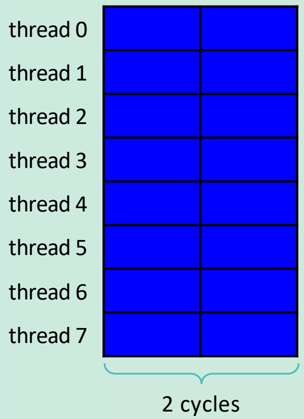
**CS 0**

```
if((threadIdx.x & 1) == 1){
  code A;
  if((threadIdx.x & 2) == 2){
    code B;
  } else{
    code C;
  }
}else{
  code D;
  if((threadIdx.x & 2) == 2){
    code E;
  } else{
    code F;
  }
}
```



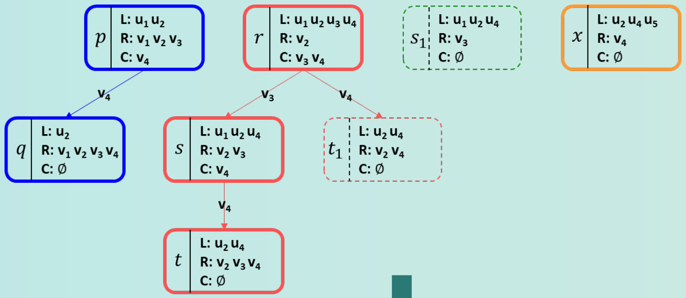

**CS 1**

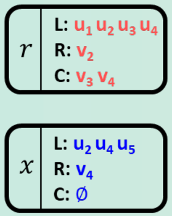
```
code A;
code A;
```




CS 0 requires more time because of the thread divergence.

# Challenge 3 : Load Imbalance

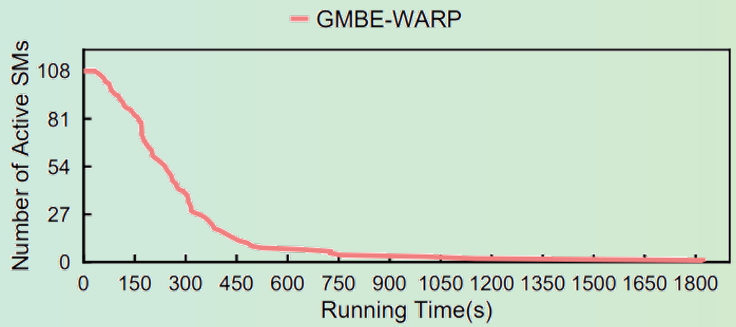



Subtrees have varying node counts

Nodes have varying vertex counts

Severe load imbalance



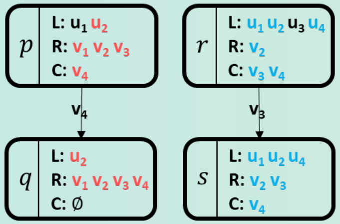

Active SMs rapidly decline over time in a straightforward algorithm.

# GMBE

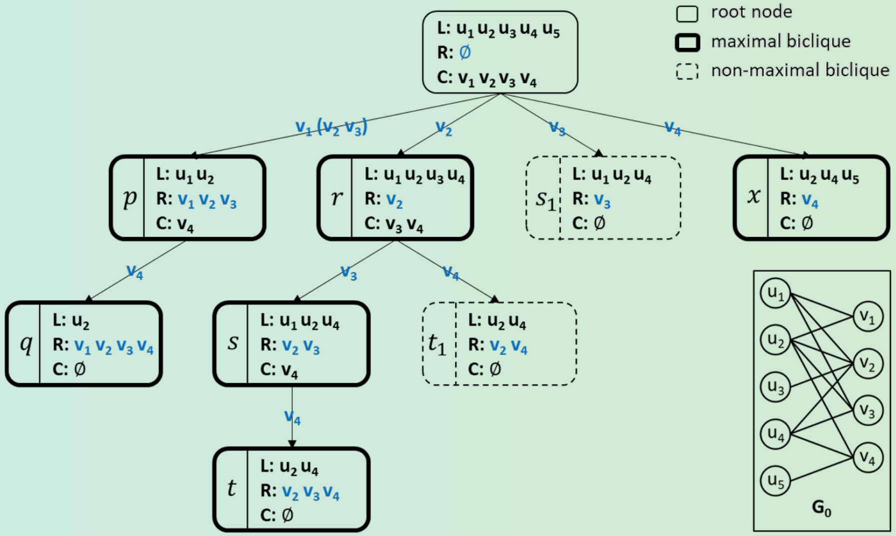
- Stack-based iteration with node reuse
- Pruning using local neighborhood sizes
- Load-aware task scheduling

# Idea 1 : Stack-based Iteration with Node Reuse

- **Key Observation**
  - Vertices in the child node are always a subset of vertices in the parent node.



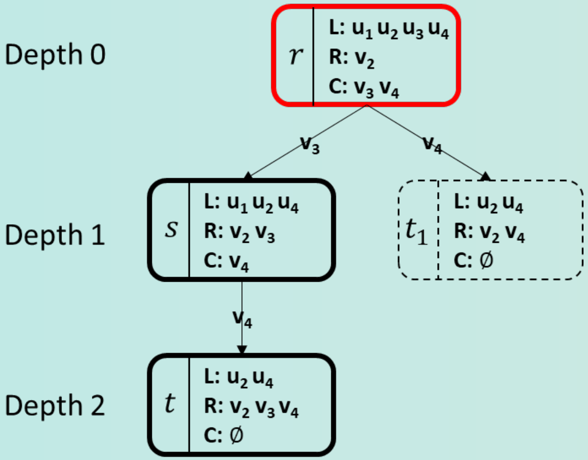

- **Main idea**
  - We allocate memory for the root node and reuse this memory to derive all nodes within the subtree, resulting in a notable reduction in memory usage.



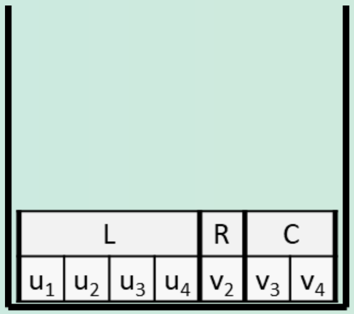
# Idea 1 : Stack-based Iteration with Node Reuse

- **Step 1 : initialize node $r$**

Enumeration tree rooted by node $r$

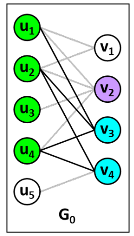



Memory usage for existing approach




Memory usage for GMBE

| | $L_r$ | | | | $R_r$ | $C_r$ | |
|---|---|---|---|---|---|---|---|
| Vertex | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $v_2$ | $v_3$ | $v_4$ |
| Depth | 0 | 0 | 0 | 0 | 0 | ∞ | ∞ |
| Local neighborhood size | | | | | | 3 | 2 |

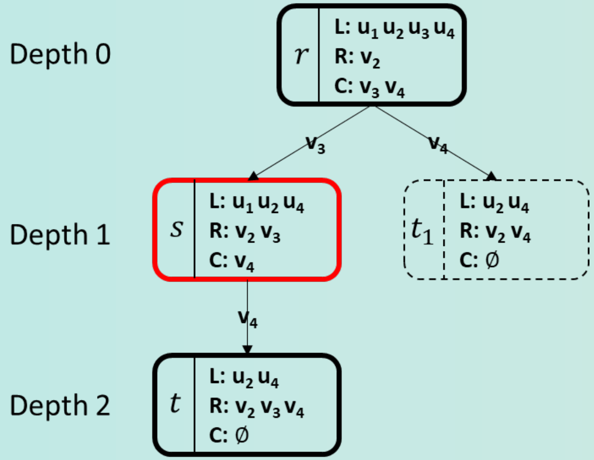



The local neighborhood size for $v_3$ is 3 because $v_3$ connect with $u_1, u_2$, and $u_4$ in $L_r$.

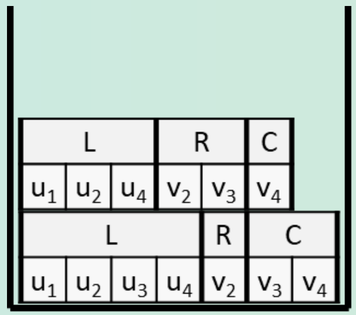
# Idea 1 : Stack-based Iteration with Node Reuse

➢ **Step 2 : generate node *s***

Enumeration tree rooted by node $r$




Memory usage for existing approach



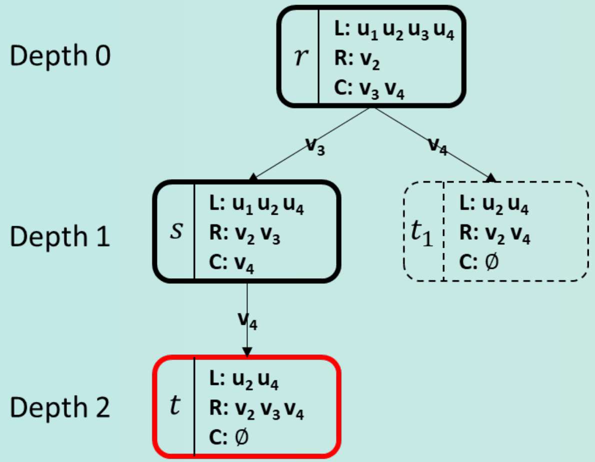

Memory usage for GMBE

| | $L_r$ | | | | $R_r$ | $C_r$ | |
|---|---|---|---|---|---|---|---|
| Vertex | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $v_2$ | $v_3$ | $v_4$ |
| Depth | **1** | **1** | 0 | **1** | 0 | **1** | ∞ |
| Local neighborhood size | | | | | | **3** | 2 |

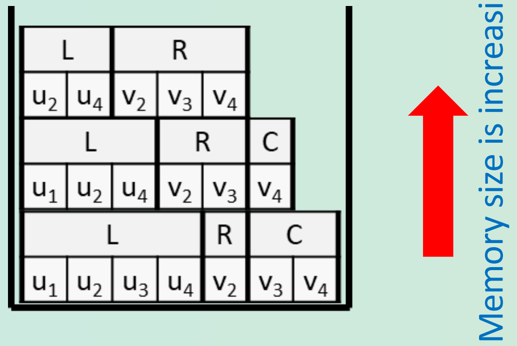
# Idea 1 : Stack-based Iteration with Node Reuse

- **Step 3 : generate node *t***

Enumeration tree rooted by node $r$




Memory usage for existing approach



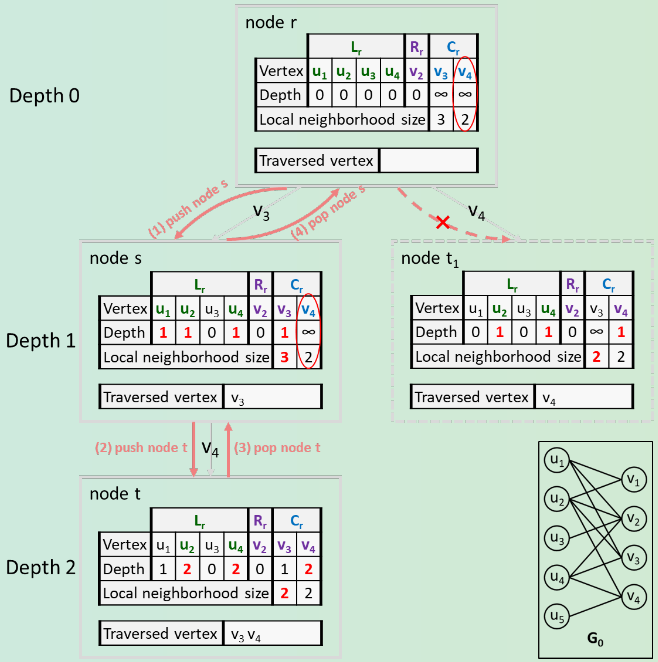

Memory usage for GMBE

| | $L_r$ | | | | $R_r$ | $C_r$ | |
|---|---|---|---|---|---|---|---|
| Vertex | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $v_2$ | $v_3$ | $v_4$ |
| Depth | 1 | **2** | 0 | **2** | 0 | 1 | **2** |
| Local neighborhood size | | | | | | **2** | 2 |

Memory size is fixed

# Idea 2 : Pruning Using Local Neighborhood Sizes

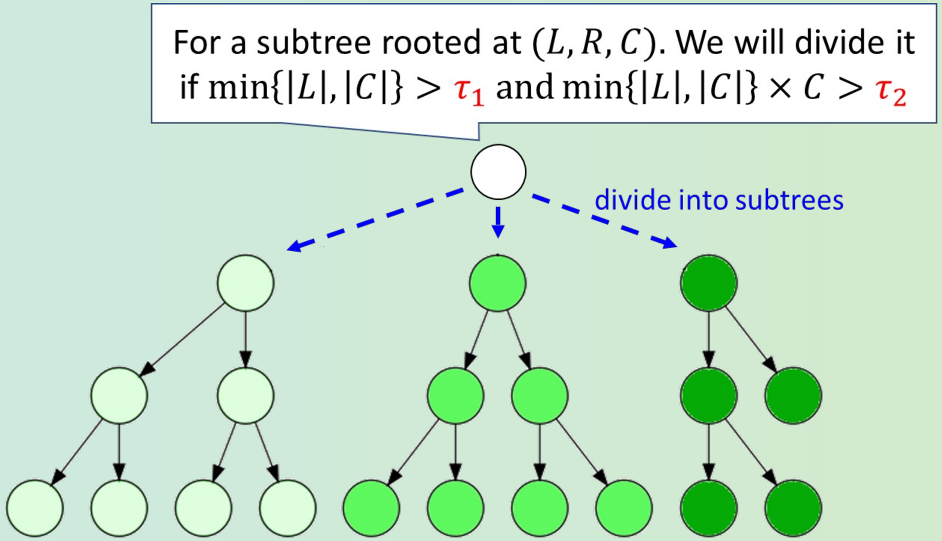
- **Key observation**
  - Local neighborhood sizes, as a necessary intermediate result, can be utilized for pruning.
- **Main idea**
  - We prune useless candidates if their local neighborhood sizes do not change after popping a traversed child node. This approach reduces thread divergence by checking multiple local neighborhood sizes simultaneously..



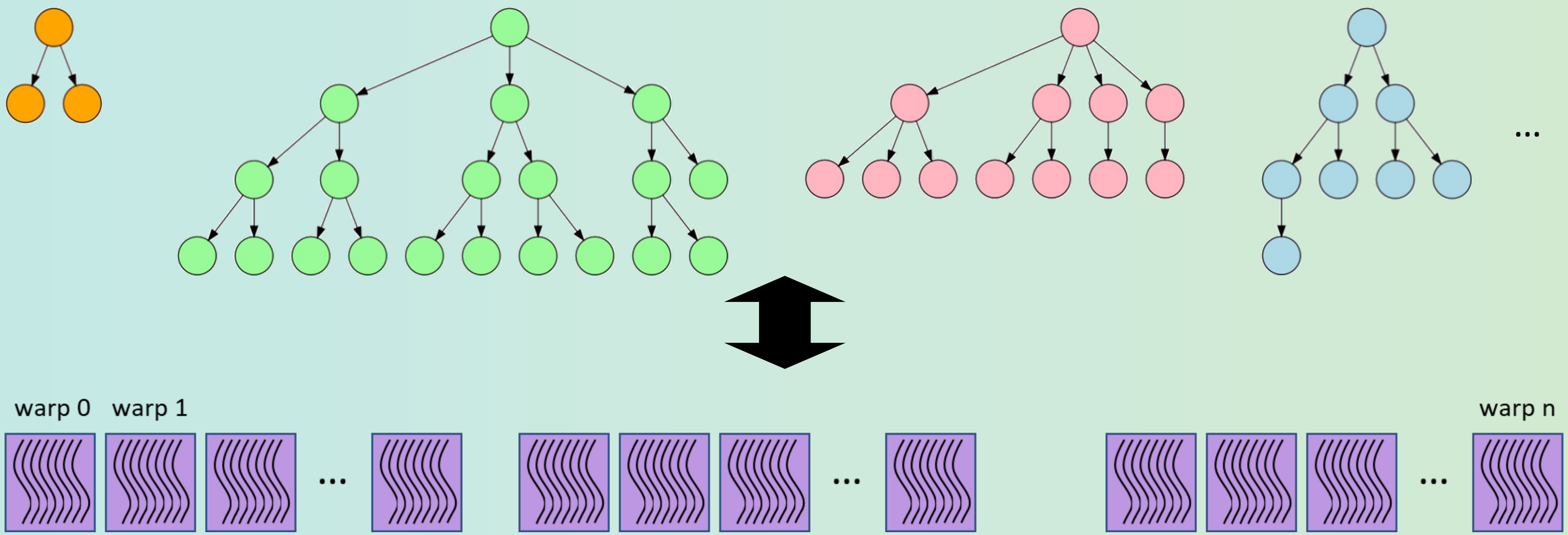

GMBE proactively prunes node $t_1$ by removing useless candidate vertex $v_4$ at node $r$ because the local neighborhood size (i.e., 2) for $v_4$ does not change after popping node $s$.

# Idea 3: Load-aware Task Scheduling

- **Key observation**
  - Due to the imbalance of subtrees in MBE problems, directly mapping subtrees to computational resources leads to significant load imbalance.
- **Main idea**
  - Design two thresholds to detect large subtree.
  - Dynamically divide large trees into multiple subtrees to balance the workloads.

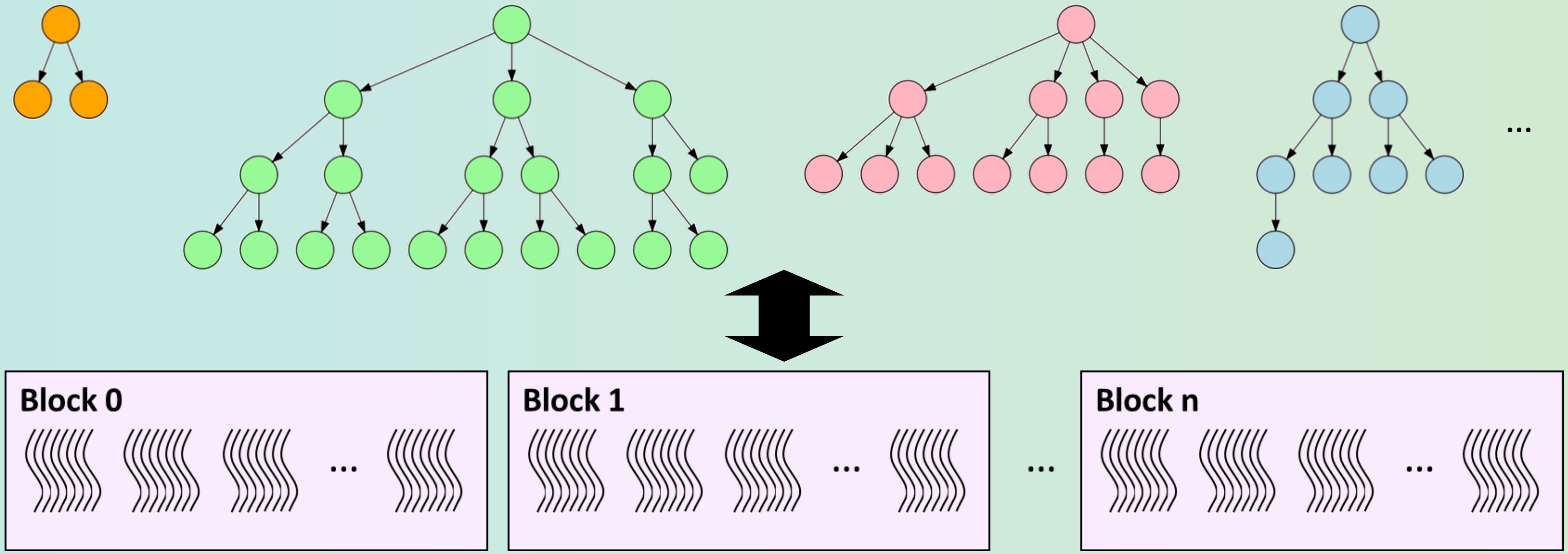


# Idea 3: Load-aware Task Scheduling



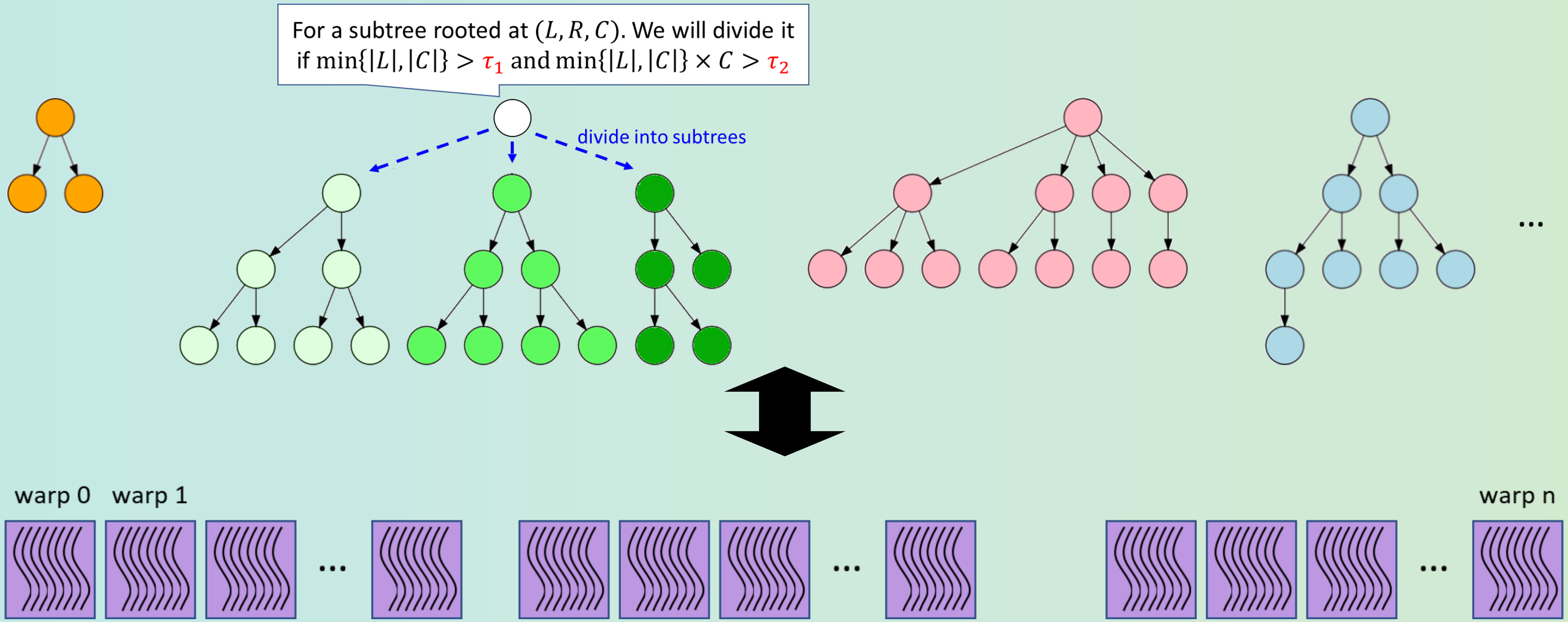

Solution 1 : Map subtrees to warps.

# Idea 3: Load-aware Task Scheduling




Solution 2 : Map subtrees to blocks.

# Idea 3: Load-aware Task Scheduling




Solution 3 : Dynamically divide large tree into multiple subtrees and map subtrees to warps.

# Evaluation

- Overall evaluation
- Effect of optimizations
- Sensitivity Analysis

# Evaluation : Overall Evaluation

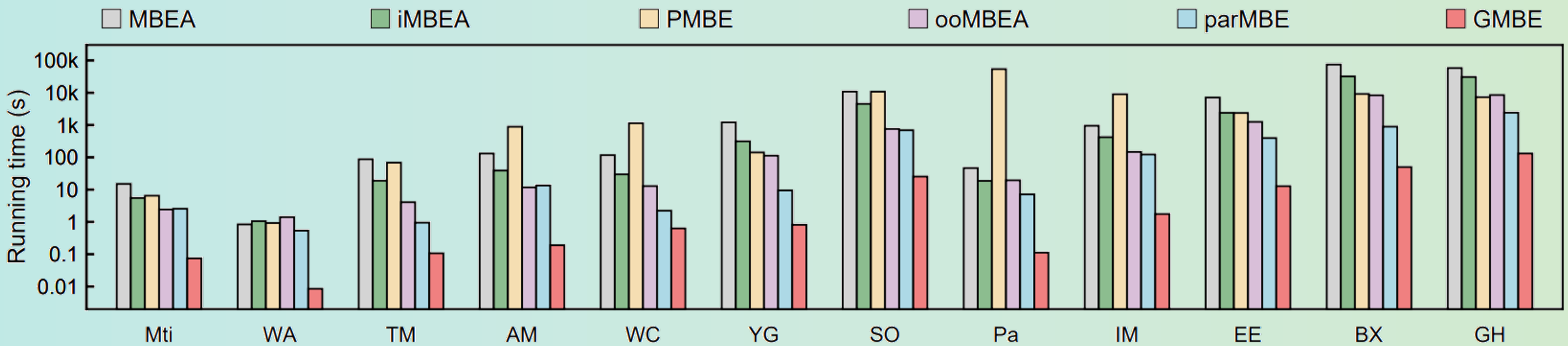



**Figure 6: Overall evaluation (log scaled).**

GMBE is 3.5×–69.8× faster than any next-best competitor on CPUs on all testing datasets.

# Evaluation : Effect of Optimizations

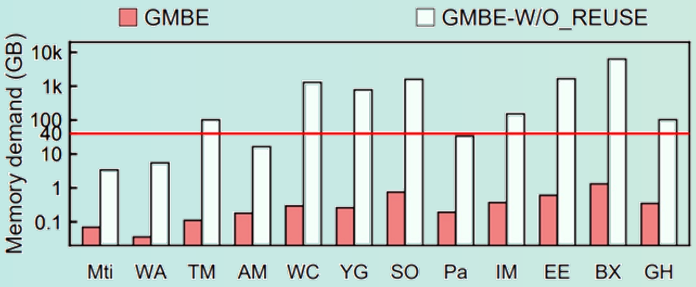



**Figure 7: Effect of the node reuse approach (log scaled).**

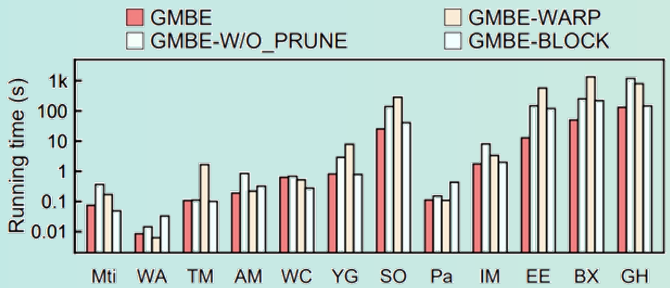



**Figure 8: Effect of pruning approach and task scheduling approach (log scaled).**

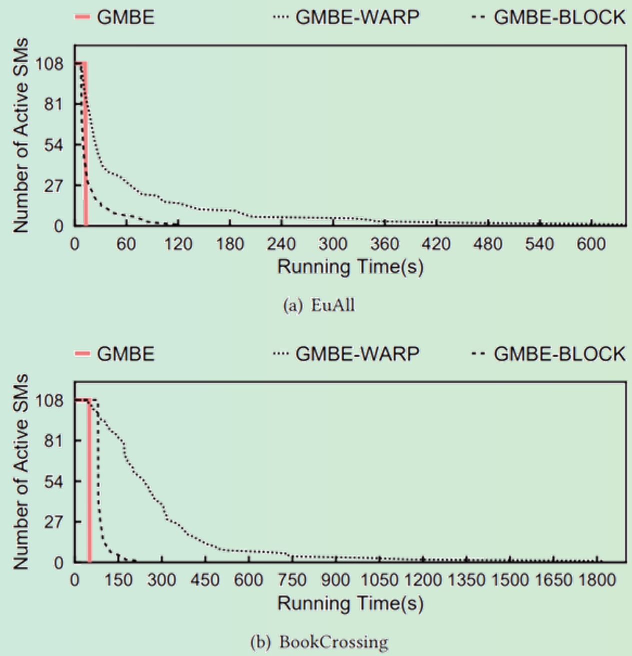
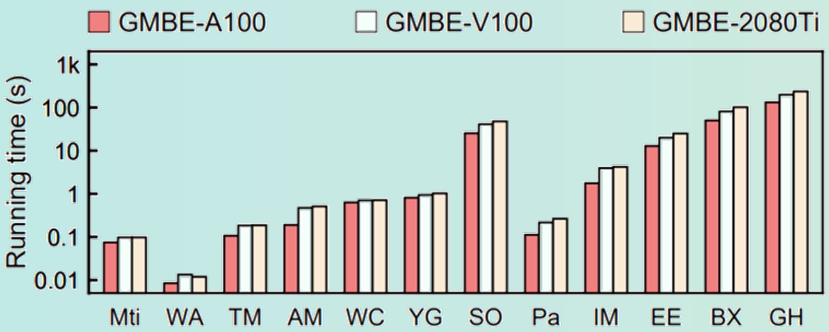


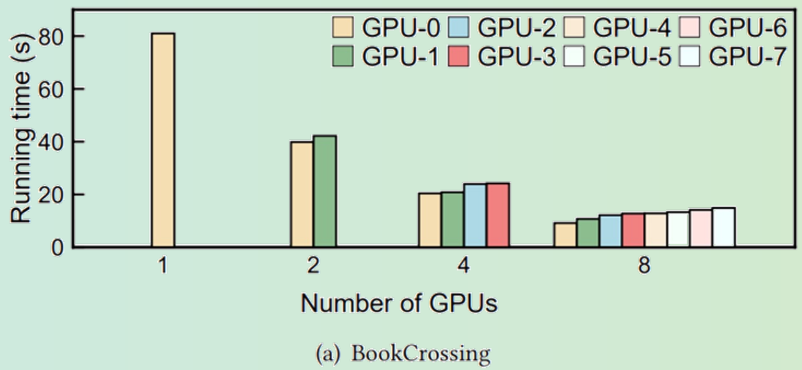

**Figure 9: Comparison of runtime loads on SMs among GMBE, GMBE-WARP, and GMBE-BLOCK.**

GMBE significantly reduces the memory usage, efficiently reduces the enumeration space, and successfully balance the workloads.

# Evaluation : Sensitivity Analysis




Adaptability on different GPU (log scaled).




Scalability of GMBE on a machine with multi-GPU.

# Q & A