Proceedings of 54th International Conference on Parallel Processing, ICPP'25, San Diego, September 8–11
University of Salerno • Department of Computer Science • Fisciano, Italy

SYgraph

A Portable Heterogeneous Graph Analytics Framework for GPUs

Antonio De Caro, Gennaro Cordasco, Biagio Cosenza
{antdecaro, gcordasco, bcosenza}@unisa.it

https://www.adecaro.eu

github.com/unisa-hpc/SYgraph

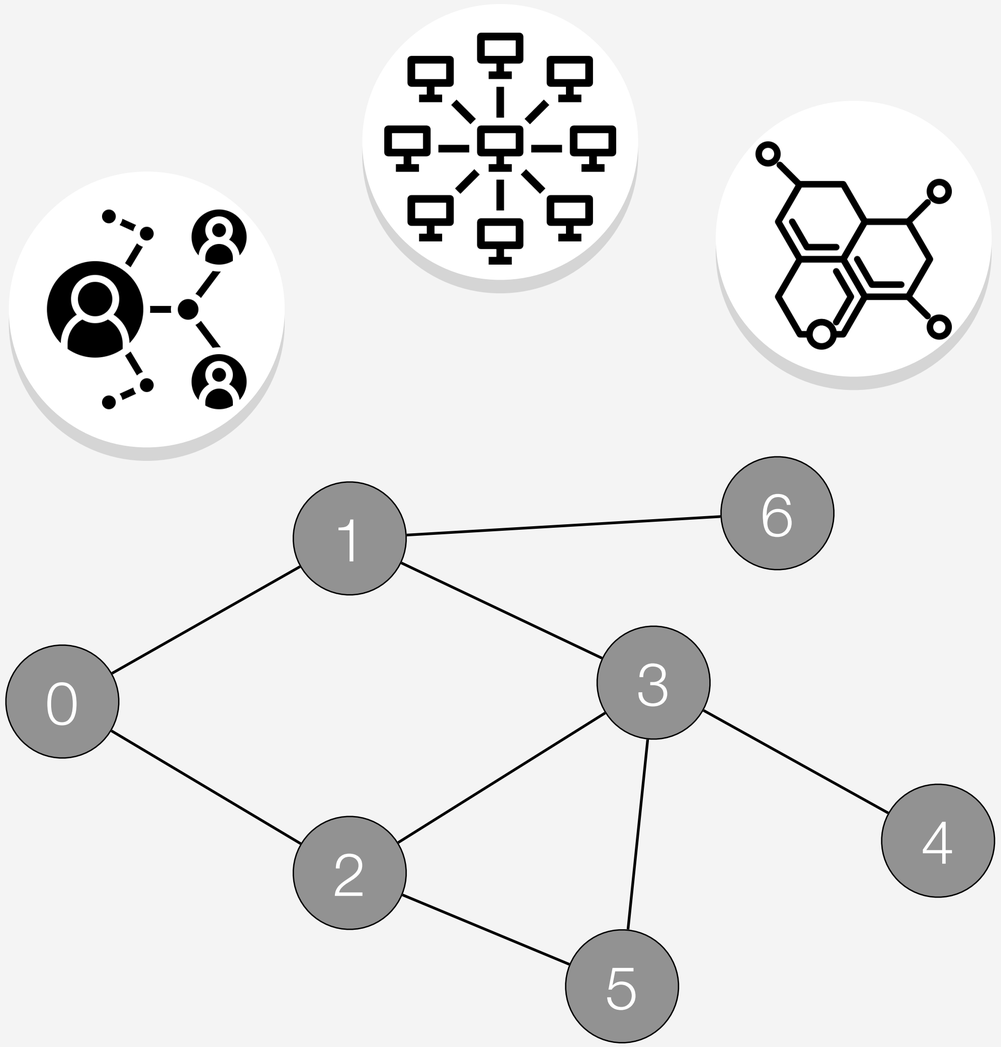
Outline

- Introduction & Background
- Contributions
  - SYgraph API
  - Execution Model
  - Two-Layer Bitmap
- Experimental Evaluation
- Conclusion

Introduction & Background

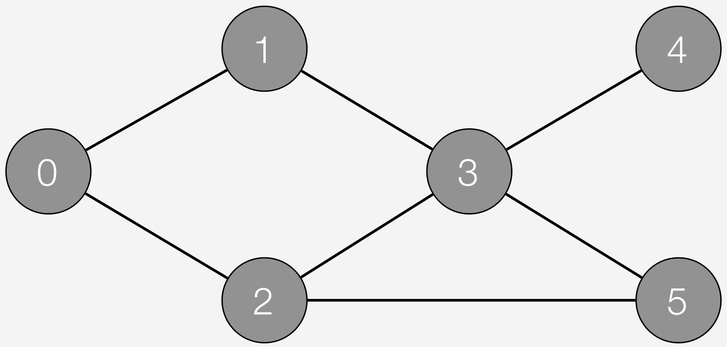
Introduction – Graph Analytics and GPUs

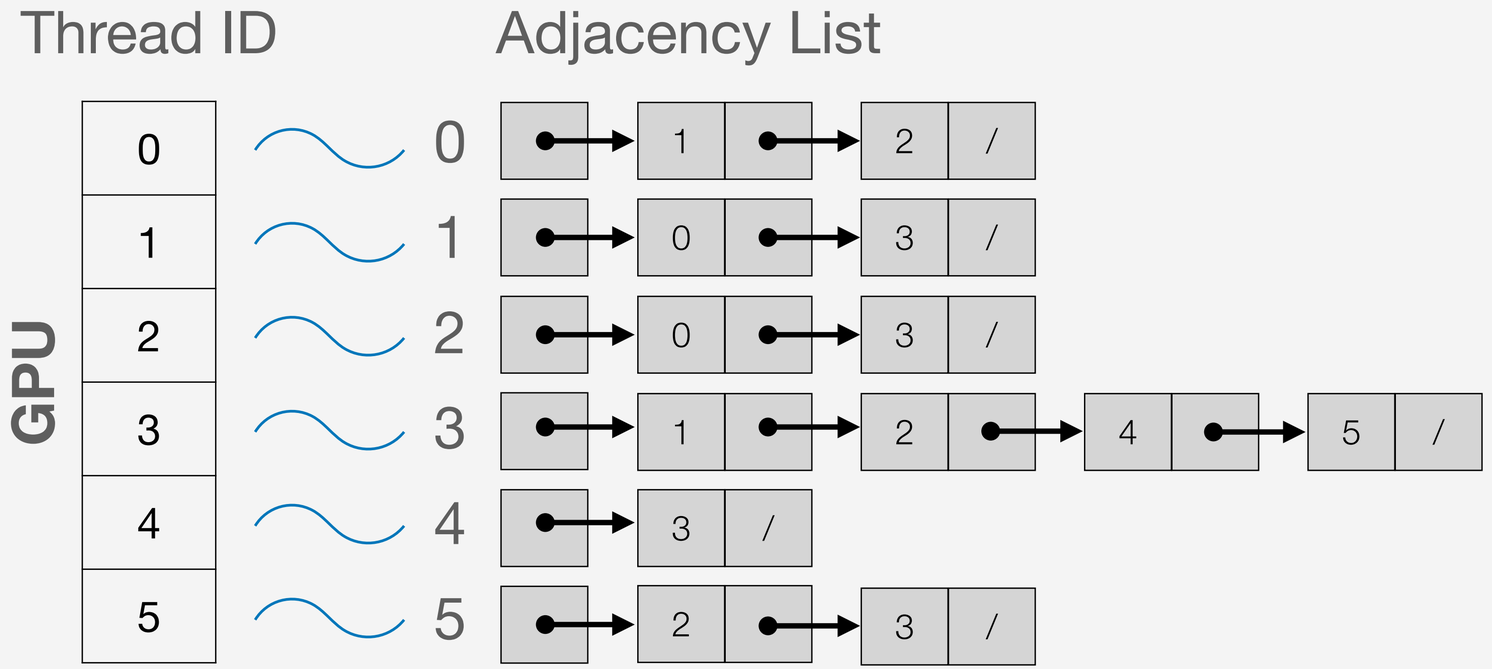
- Many **real-world applications** are naturally **modeled as graphs**
  - Social networks (e.g., friend/follow graphs)
  - Web and hyperlinks (e.g., crawling, ranking)
  - and more …
- These graphs are **huge**: millions of nodes, billions of edges
- **GPUs** offer **thousands of cores** that can process nodes and edges in **parallel**



Introduction – Why are Graphs Challenging on GPUs?

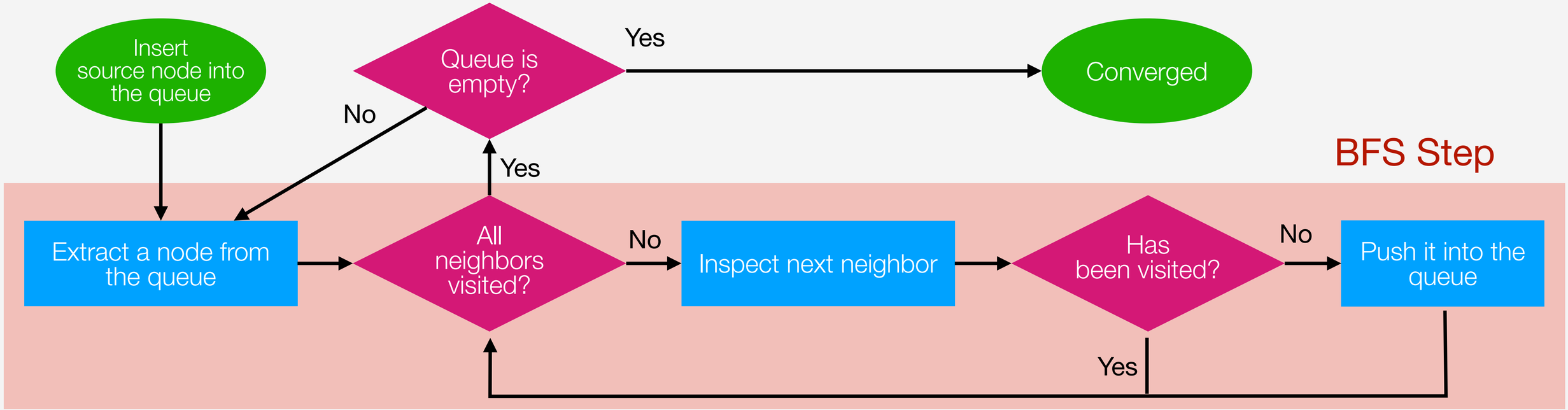
- **GPUs** struggle to handle **irregular** workloads such as **graph traversal** due to:
  - Skewed degree distribution
  - Irregular topology
  - Memory-boundness





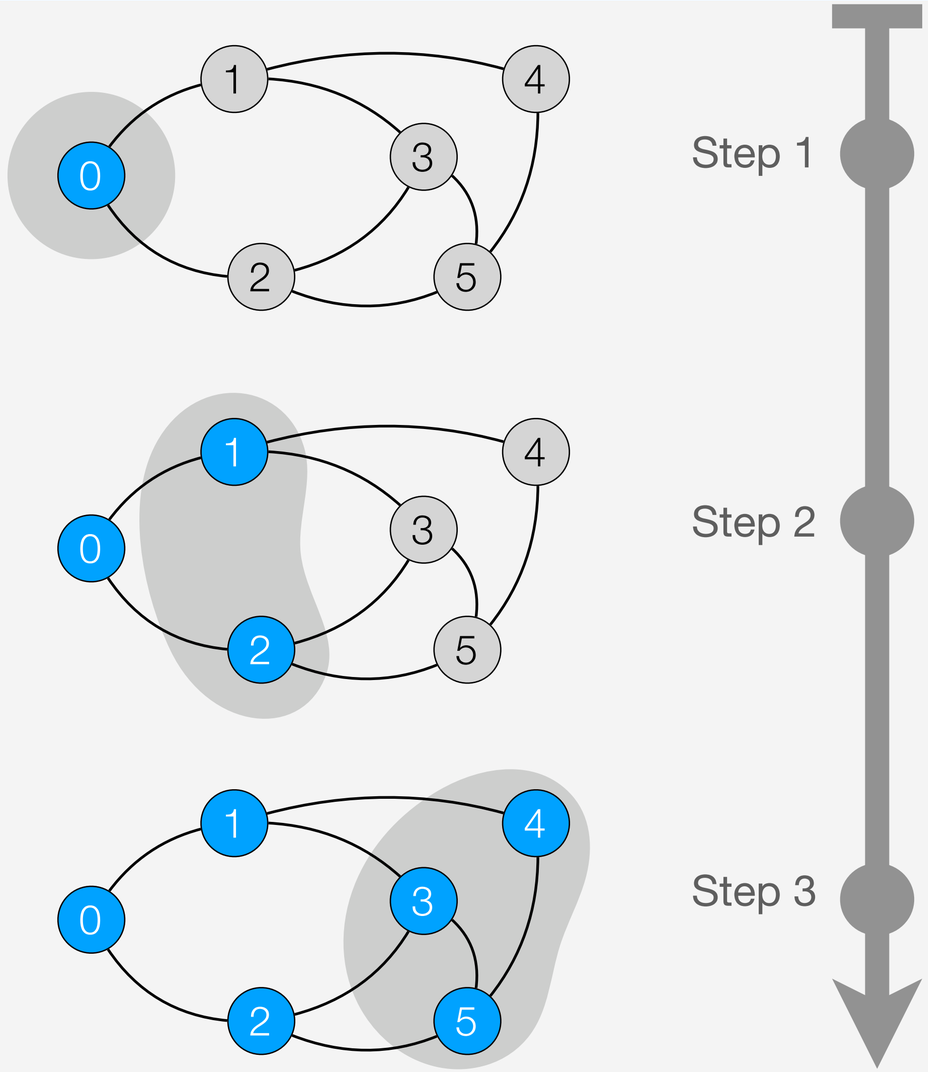
Background – Graph Algorithms as Iterative Converging Processes

- Most graph algorithms are **iterative**;
- They progress step-by-step until a **convergence condition** is met.
- An example: Breadth First Search (BFS)



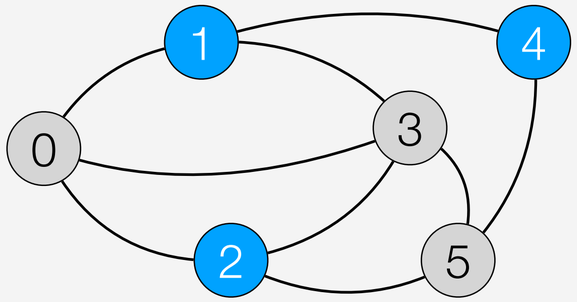
Background – Frontier-Based Processing

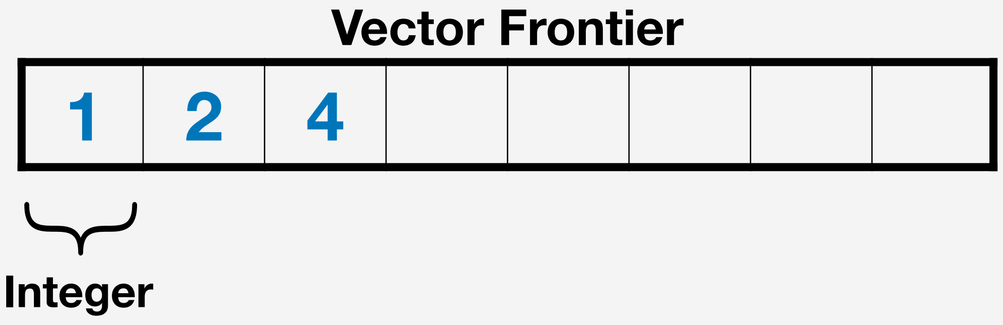
- In parallel graph algorithms, a **Frontier** is the set of active nodes during an iteration
- Example: *in BFS, it holds nodes at current distance level*
- Algorithms proceed in supersteps:
  1. Expand frontier
  2. Apply logic
  3. Prepare next iteration

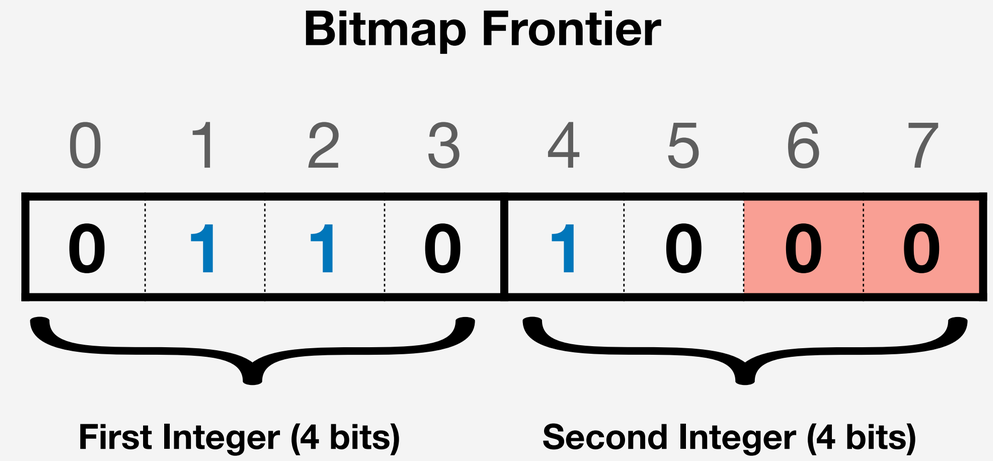


Background – Frontier Representation

- **Vector-based** frontier:
  - Each entry represents a node in the frontier
- **Bitmap-based** frontier:
  - Each bit i represents whether the vertex i is in the frontier

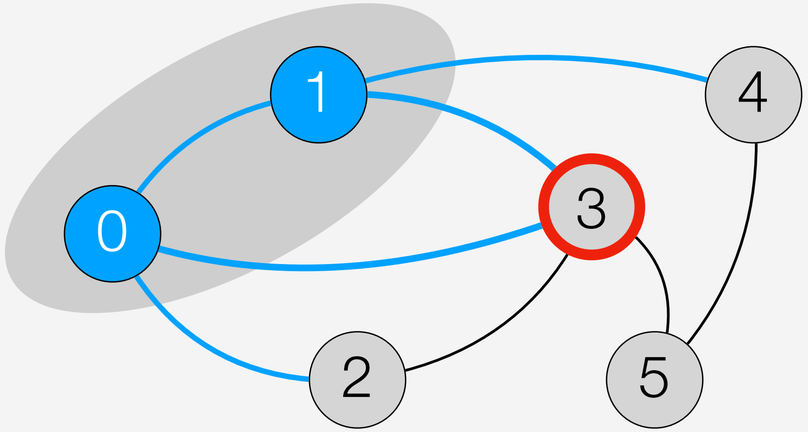


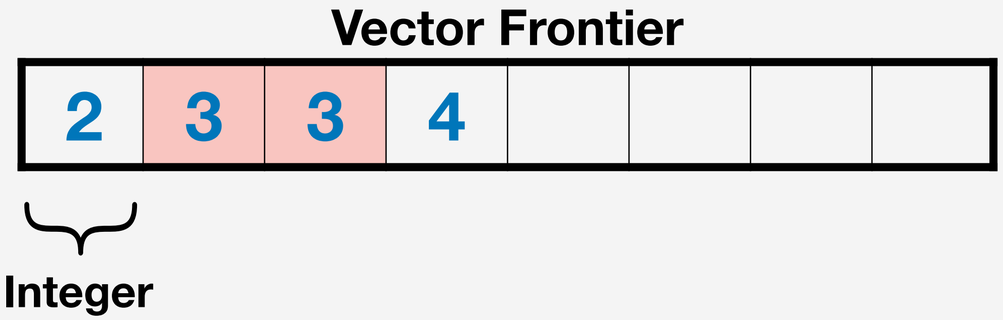


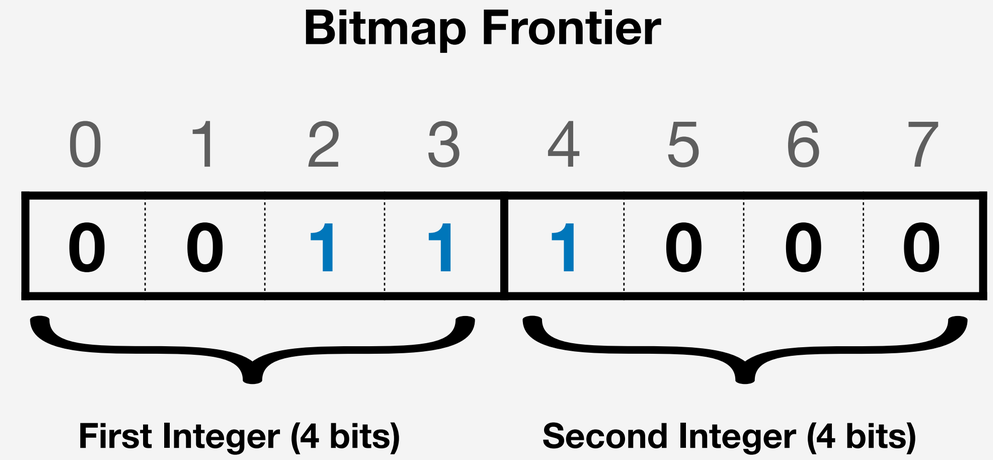


Background – Bitmap Frontier vs. Vector Frontier

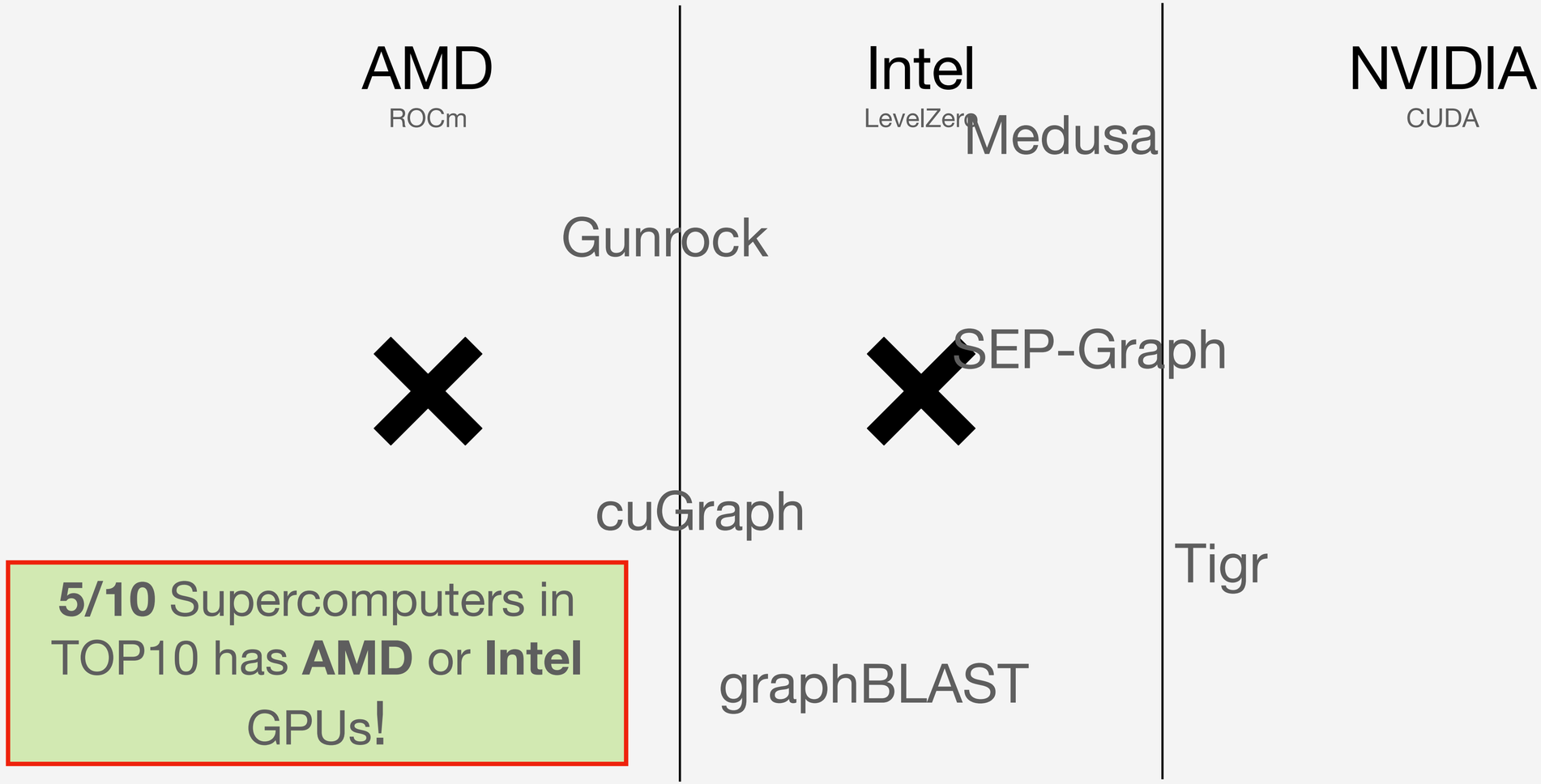
- Bitmap has no duplicated nodes for advance operations
  - Which means no search after the advance operation
- Less space for scale-free graphs!







Background – SOTA GPU Graph Frameworks



5/10 Supercomputers in TOP10 has **AMD** or **Intel** GPUs!

Contributions

Contributions

1. **SYgraph**, the first **portable**, **heterogeneous** graph analytics GPU **framework** based on SYCL and C++20

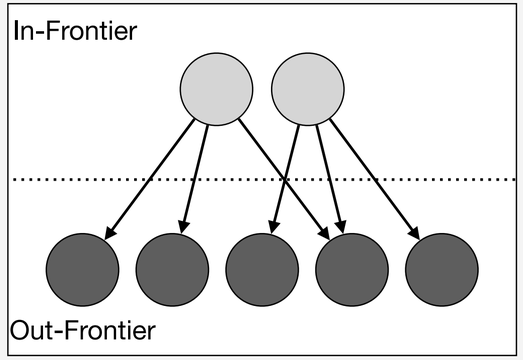
2. A **load-balanced execution model** for graph traversal

3. A novel memory-efficient **Two-Layer Bitmap Frontier**

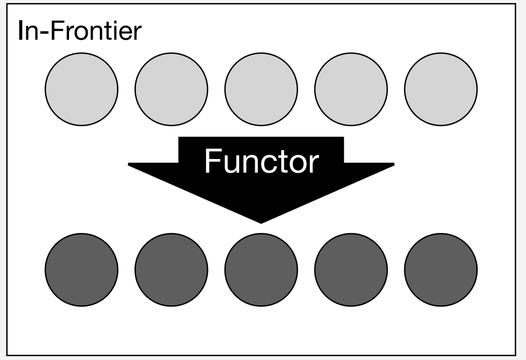
SYgraph API

SYgraph API — Primitives

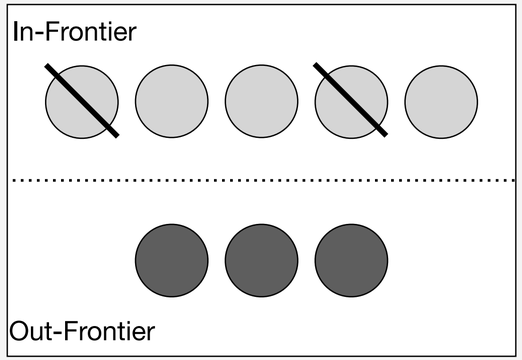
- SYgraph provides fundamental building blocks called **primitives**
- There are three core primitives:
  - **Advance** (Edge Operation)
  - **Compute** (Vertex Operation)
  - **Filter**
- Each primitive uses a user-defined lambda function that enables the data-driven behavior



Advance



Compute



Filter

SYgraph API — An Example of BFS

- SYgraph API Components:
  - **Primitives**: apply user-defined logic
  - **Frontier**: handles dynamic sets of active elements
  - **Graph**: common graph operations
  - **I/O**: manages read/write operations of graphs
- In yellow the code executed on the GPU
- Each lambda captures the pointer to data structures allocated by the host, such as the `dist` array

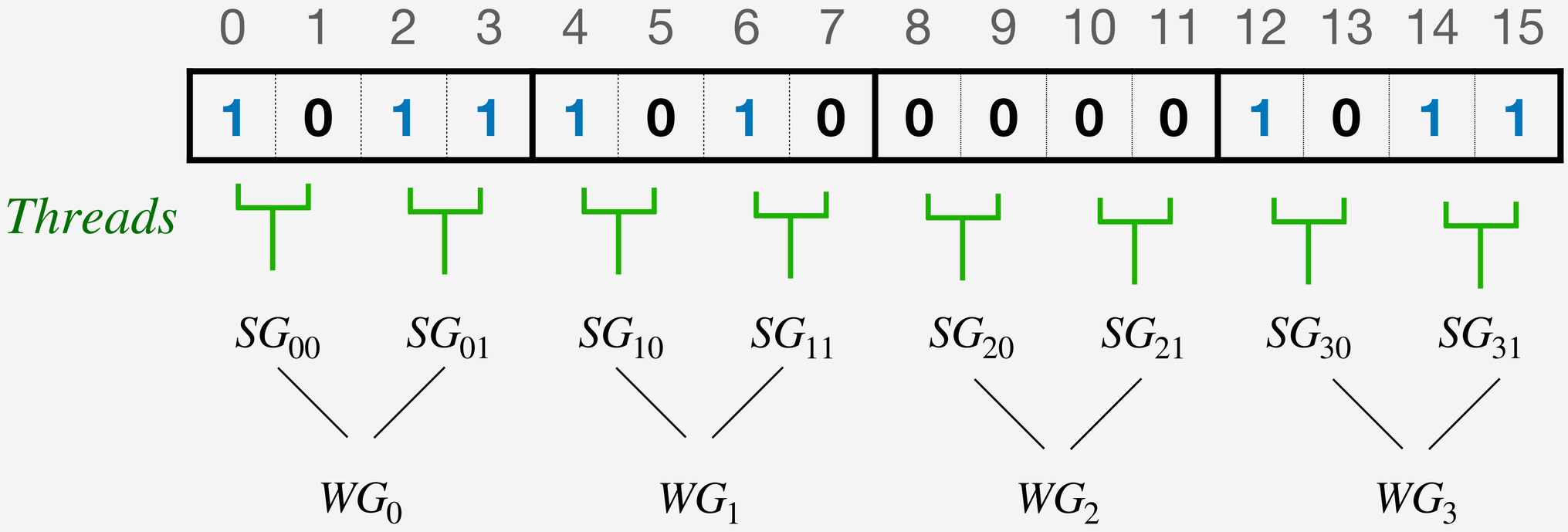
```
1.  using namespace sygraph;
2.  void BFS(Graph& G, int* dist, vertex_t src){
3.  auto in_frontier  = makeFrontier(G)
4.  auto out_frontier = makeFrontier(G)
5.  in_frontier.insert(src)
6.  size_t size = G.getVertexCount()
7.  int iter = 0
8.  while (!in_frontier.empty()) {
9.  operators::advance::frontier(G, in_frontier, out_frontier,
10.   [=](vertex_t u, vertex_t v, edge_t e, weight_t w) {
11.     bool visited = dist[v] < (size + 1)
12.     return !visited
13.   }).wait()
14. operators::compute::execute(G, out_frontier,
15.   [=](vertex_t v) {
16.     dist[v] = iter + 1
17.   }).wait()
18. frontier::swap(in_frontier, out_frontier)
19. out_frontier.clear()
20. iter++
21. }}
```

Execution Model

Execution Model

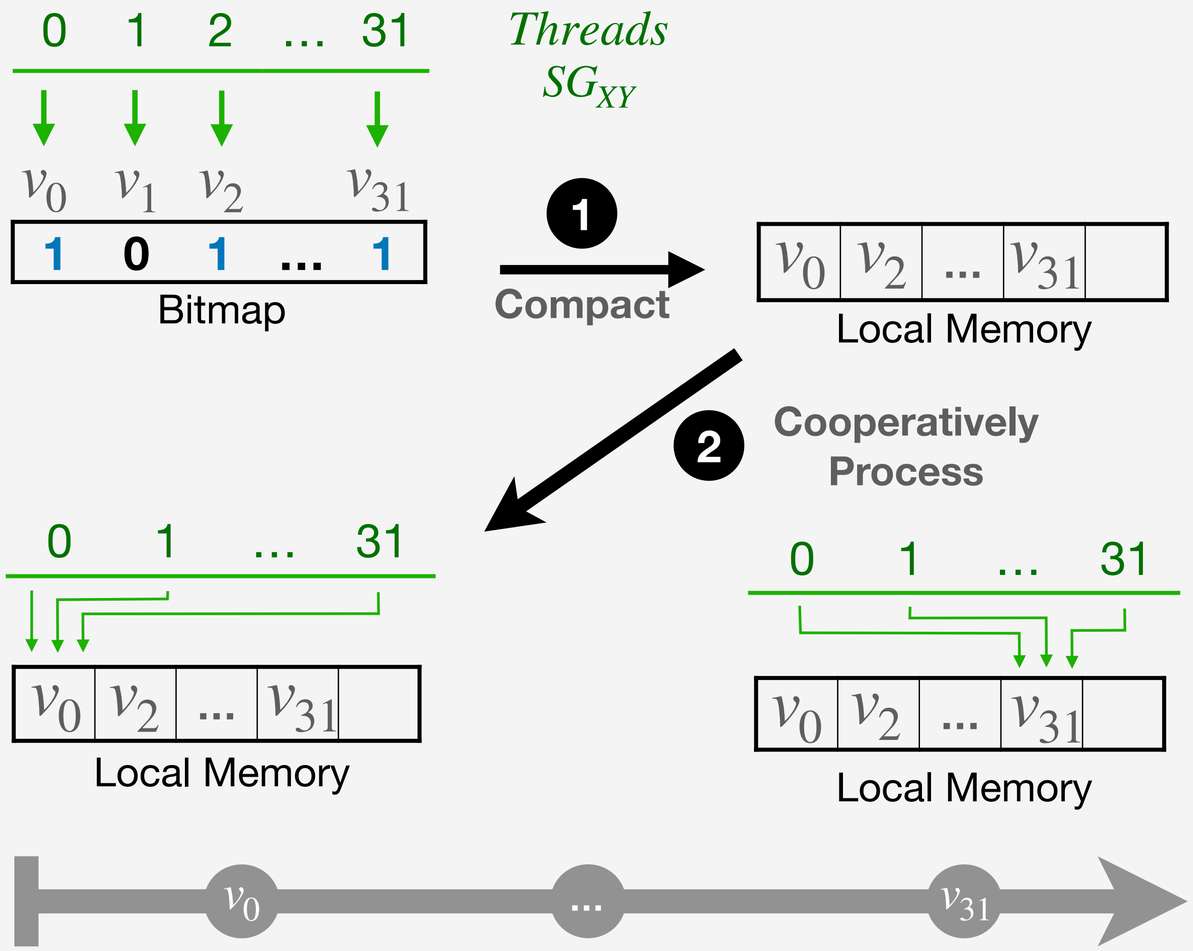
- SYgraph uses a **bitmap-based** frontier representation

- Work-group (WG): like CUDA Block
- Sub-group (SG): like CUDA Warp



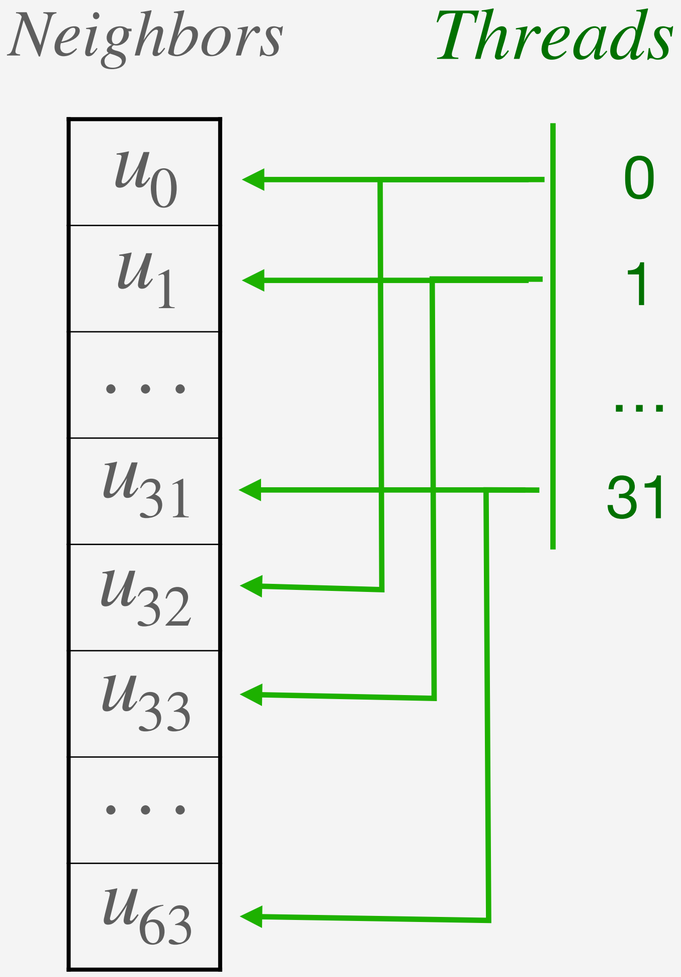
Execution Model – Subgroup Workload during Advance

1. Initially, **each thread** is assigned to a **single bit** (i.e. vertex)
2. With a sub-group **scan operation**, the threads compact the active vertices into **local memory**
3. The threads then begin processing the neighborhood of each active vertex in a **cooperative fashion**
4. This process ends when all the active vertices in the current Integer are visited



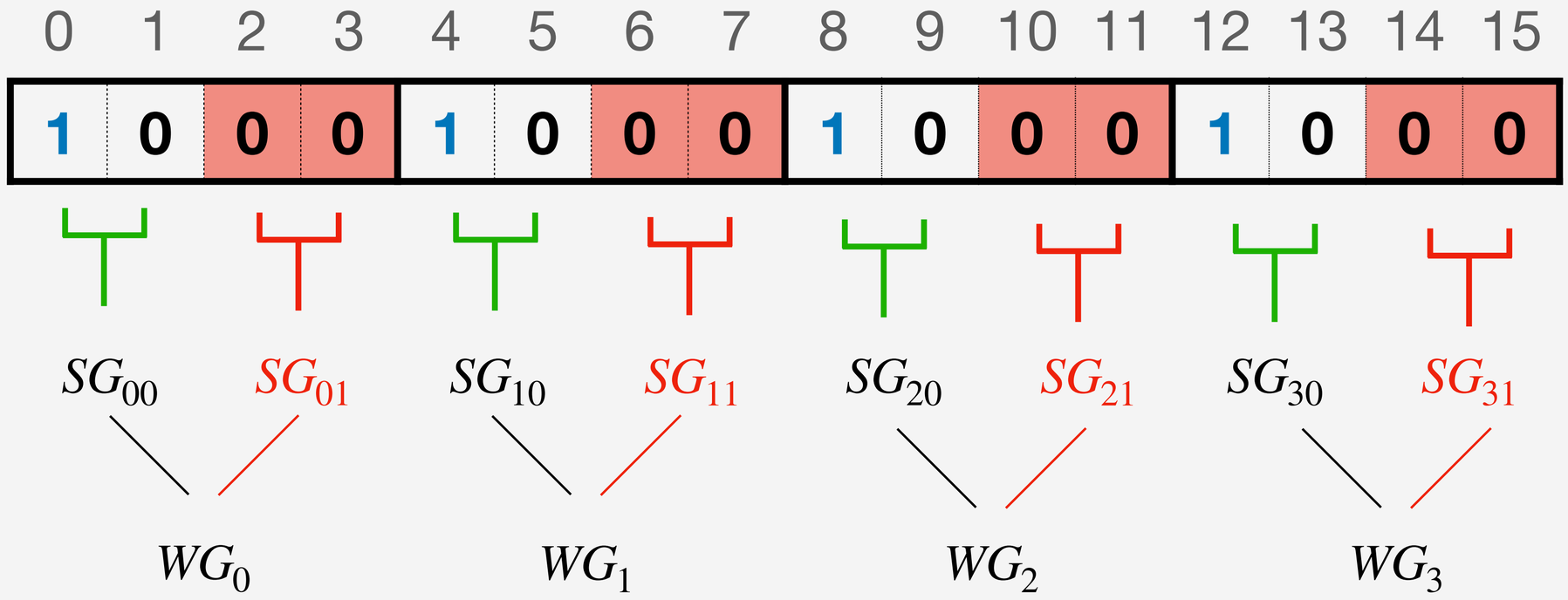
Execution Model – Cooperative Processing

- Each **thread** is assigned to a **different neighbor** of a specific active vertex
- During this step, each thread applies the **user-defined lambda** to determine if the neighbor should be added to the output frontier
- Access to the neighborhood occurs in a **coalesced fashion**
- This process concludes when all neighbors have been evaluated



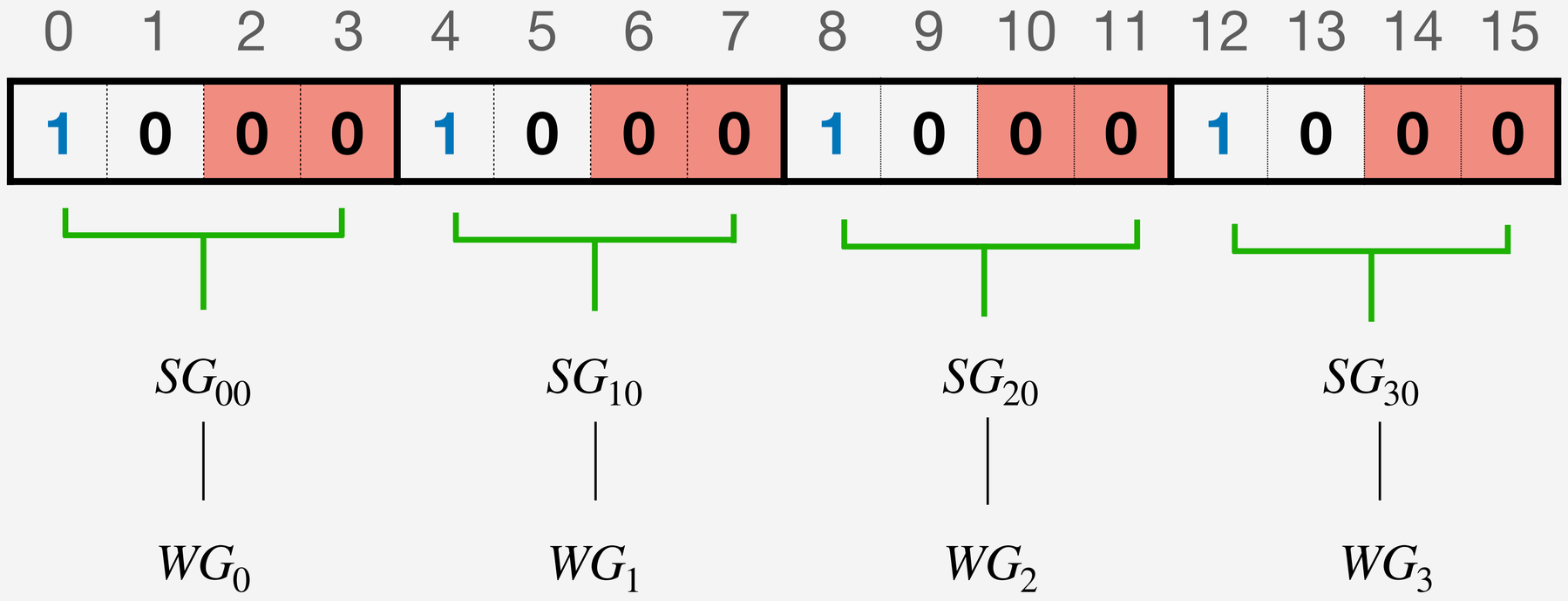
Bitmap Frontier – A problem

- What happens if only one bit is set to **1** for each Integer?



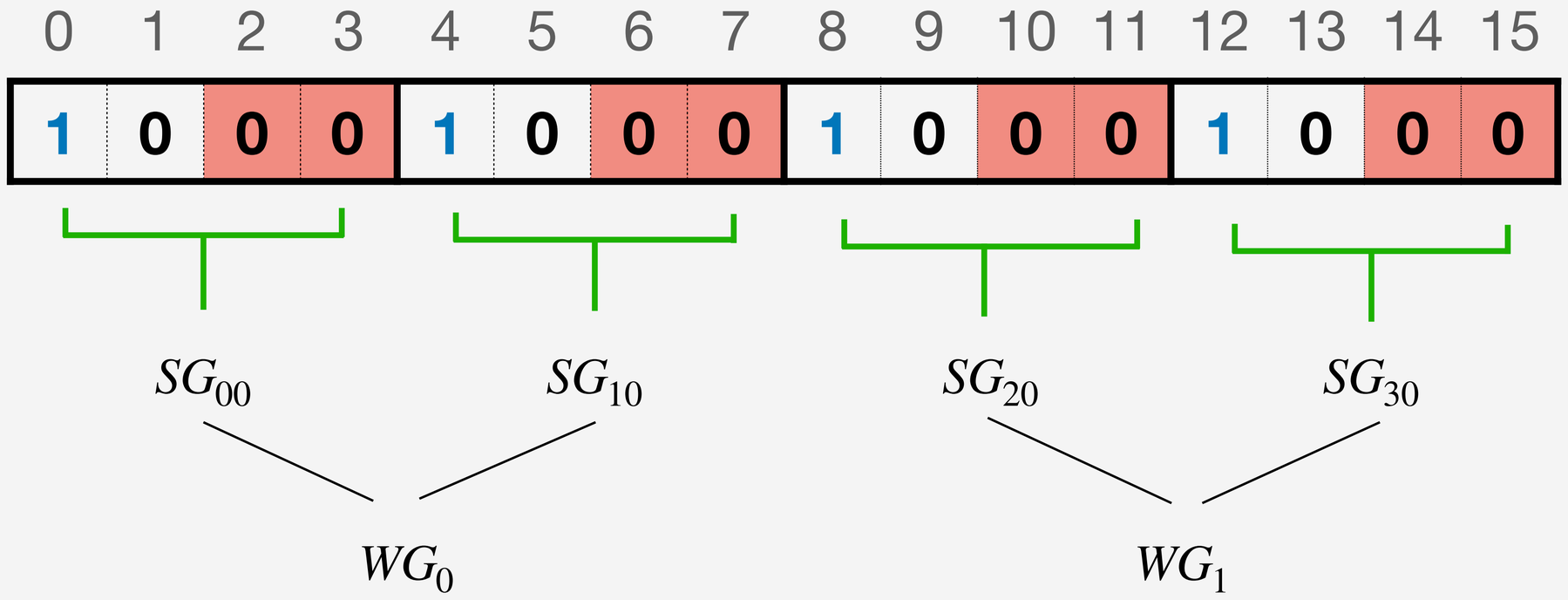
Bitmap Frontier – A solution

- Adjust the **bitmap Integer size** to **match** the **subgroup size**
- All threads within a sub-group will stay active during the neighborhood processing



Bitmap Frontier – A solution

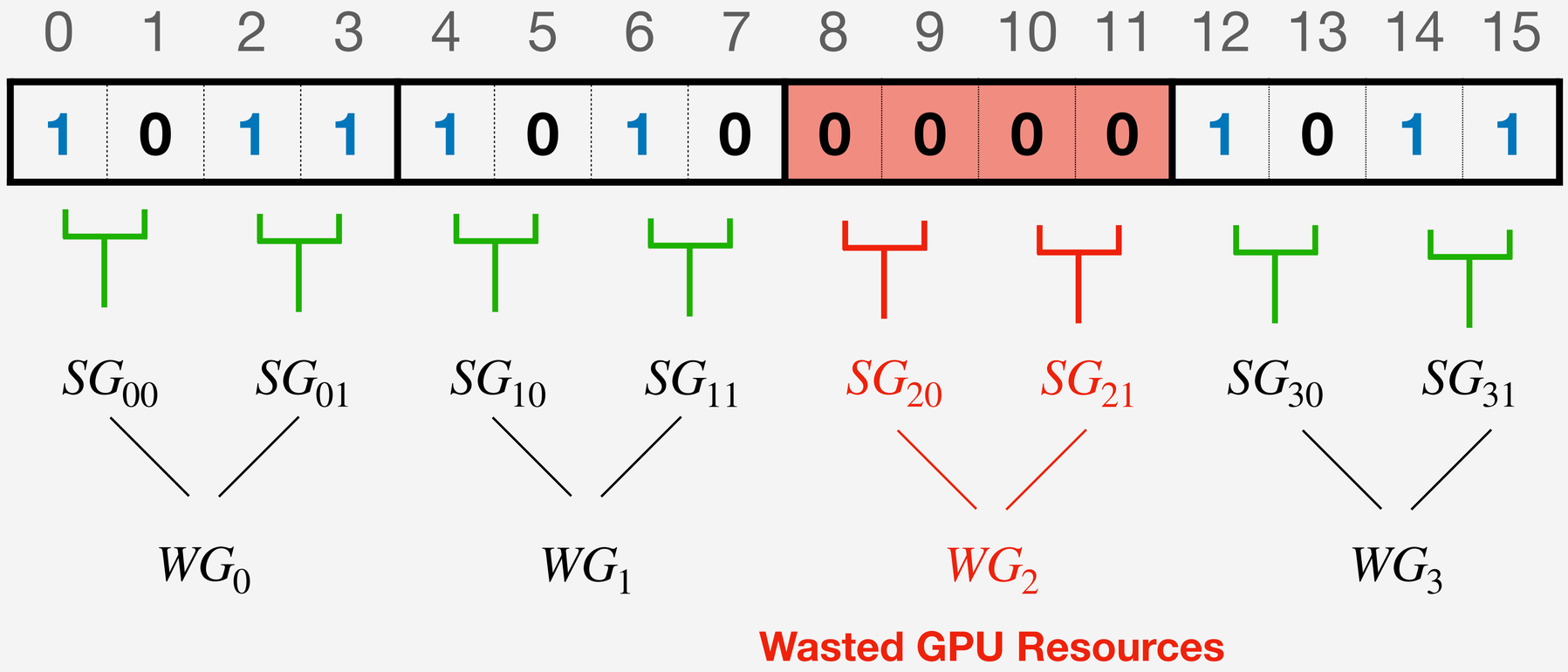
- Apply a **coarsening factor** to improve GPU resource usage
- Each work-group will be assigned to multiple bitmap Integers



Two-Layer Bitmap

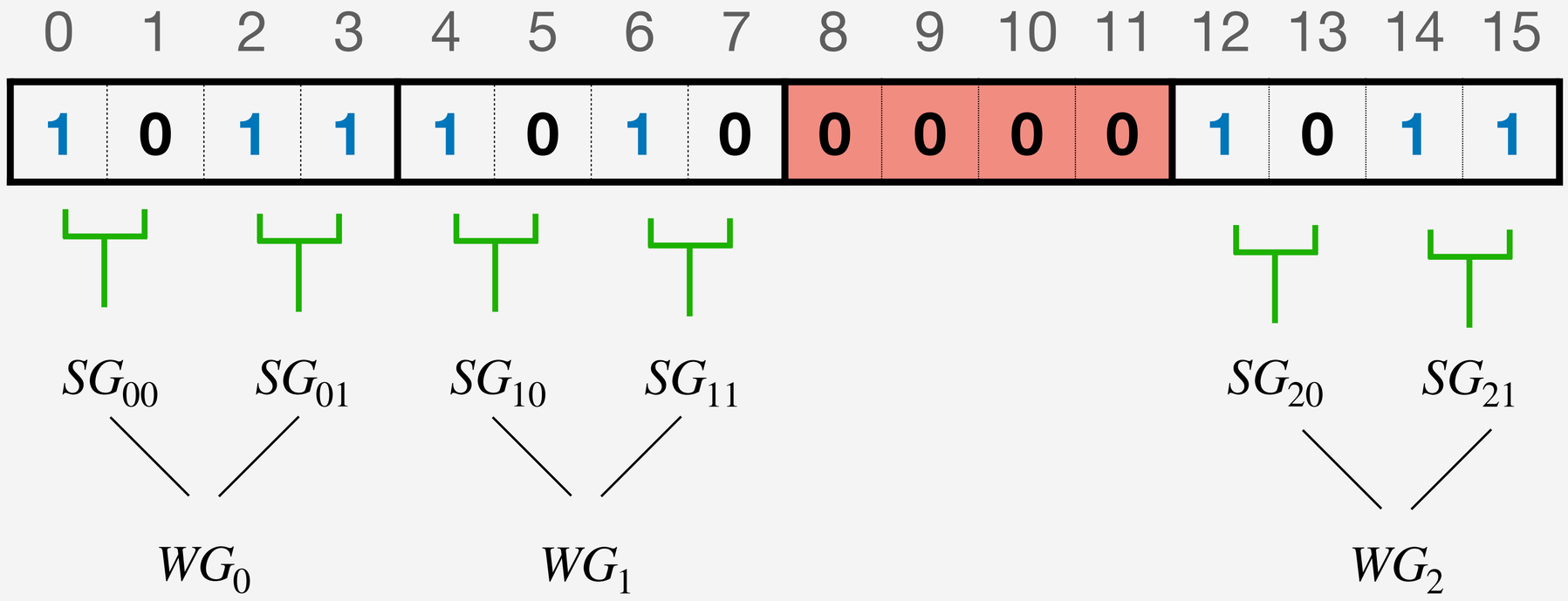
Bitmap Frontier – Another problem

- What happens when an **Integer** is **0**?



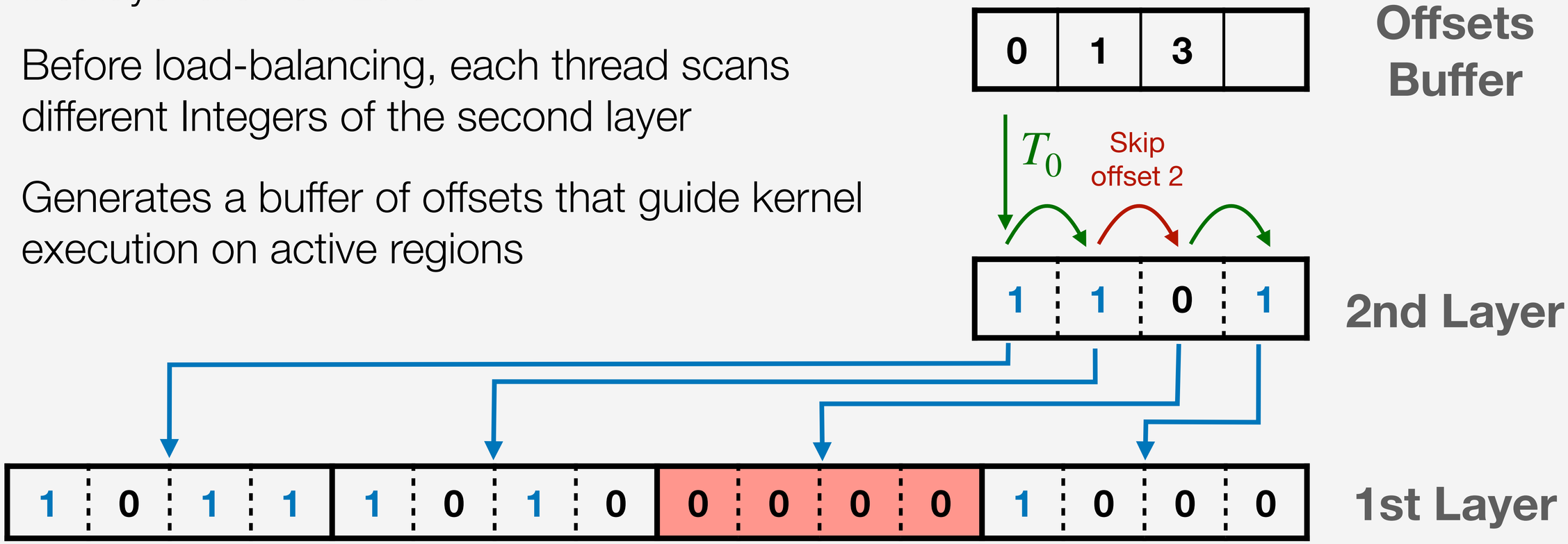
Bitmap Frontier – Another solution

- Do not assign a work-group to that Integer. How?



Two-Layer Bitmap

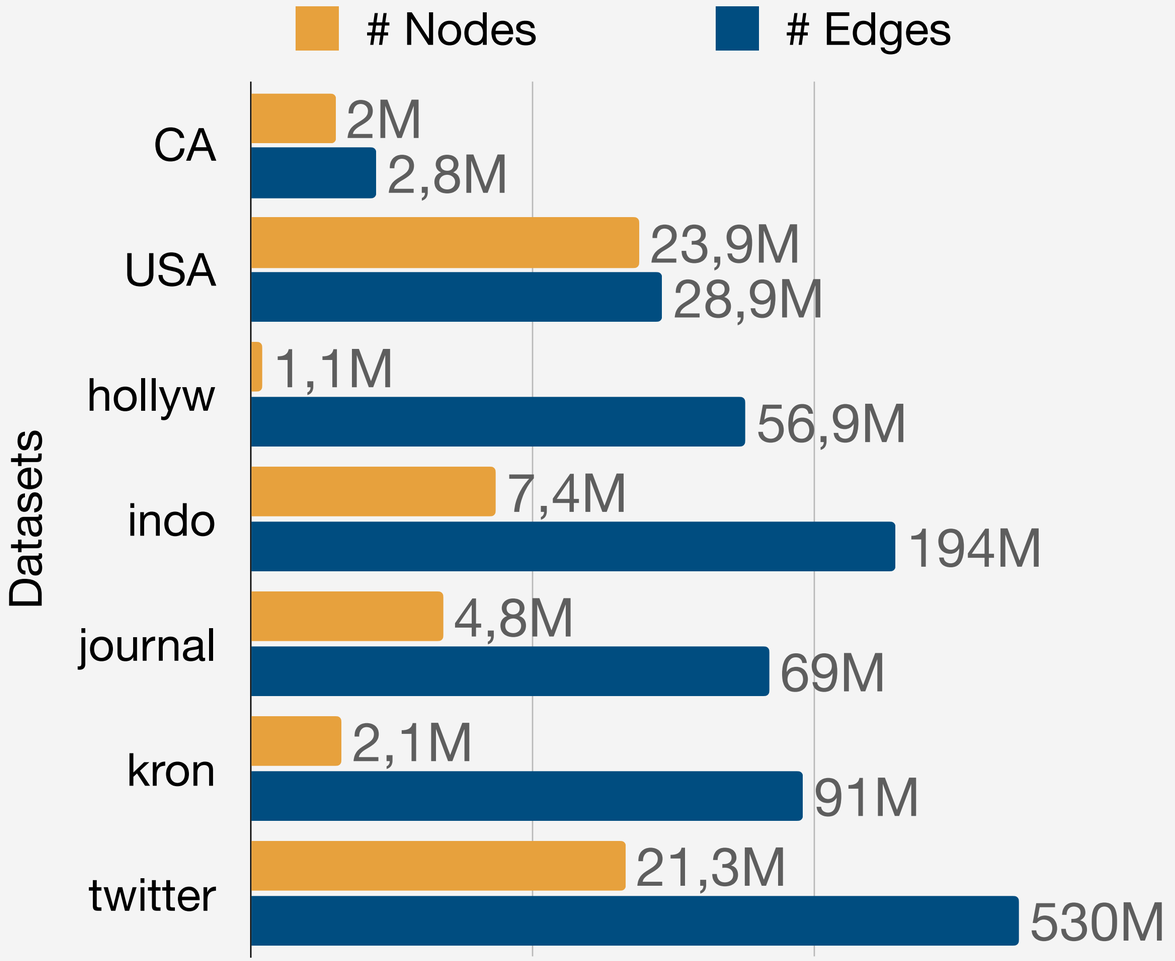
- The second layer marks which Integers in the first layer are non-zero
- Before load-balancing, each thread scans different Integers of the second layer
- Generates a buffer of offsets that guide kernel execution on active regions



Experimental Evaluation

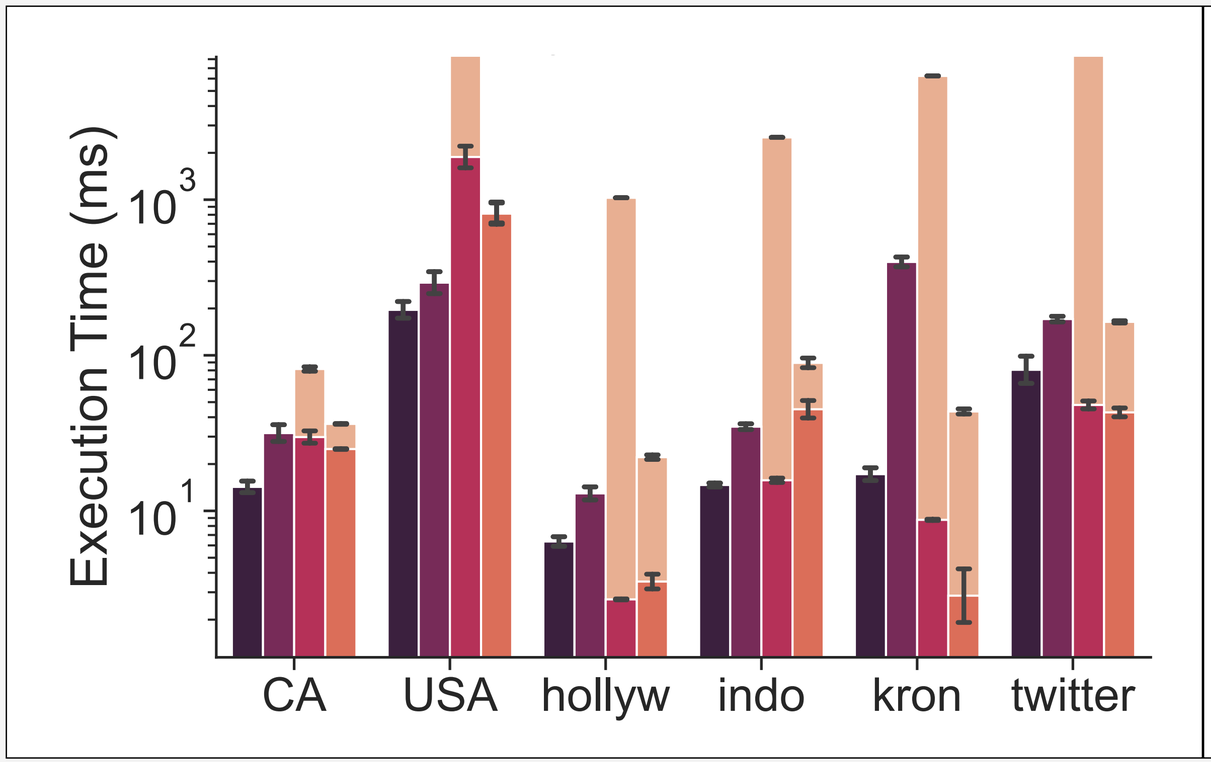
Experimental Evaluation — Methodology

- We compared SYgraph against:
  - Gunrock
  - Tigr
  - SEP-Graph
- Experiments against state-of-the-art were conducted on a **NVIDIA V100S**
- **High-diameter** and **Scale-free** graphs:
  - From 2 to 24 Millions nodes
  - From 2,1 to 530 Millions edges

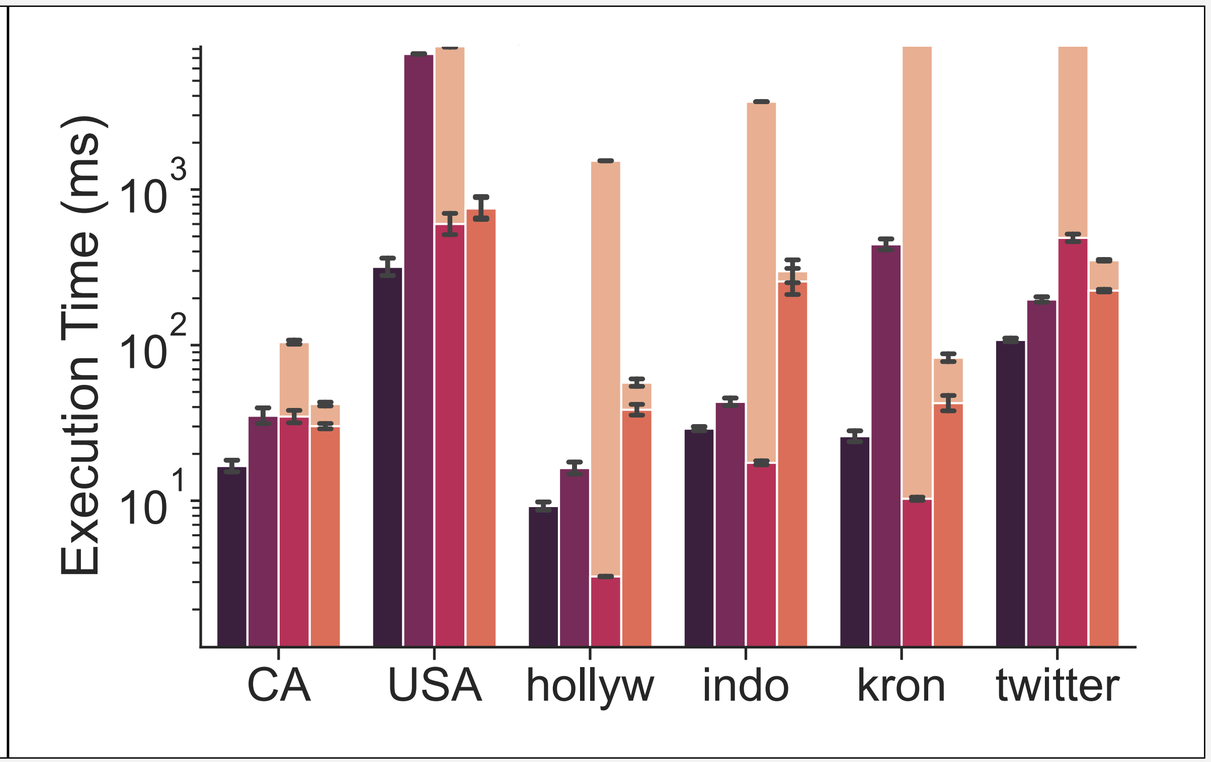


Experimental Evaluation — Performance vs. SOTA Frameworks

Breadth First Search (BFS)

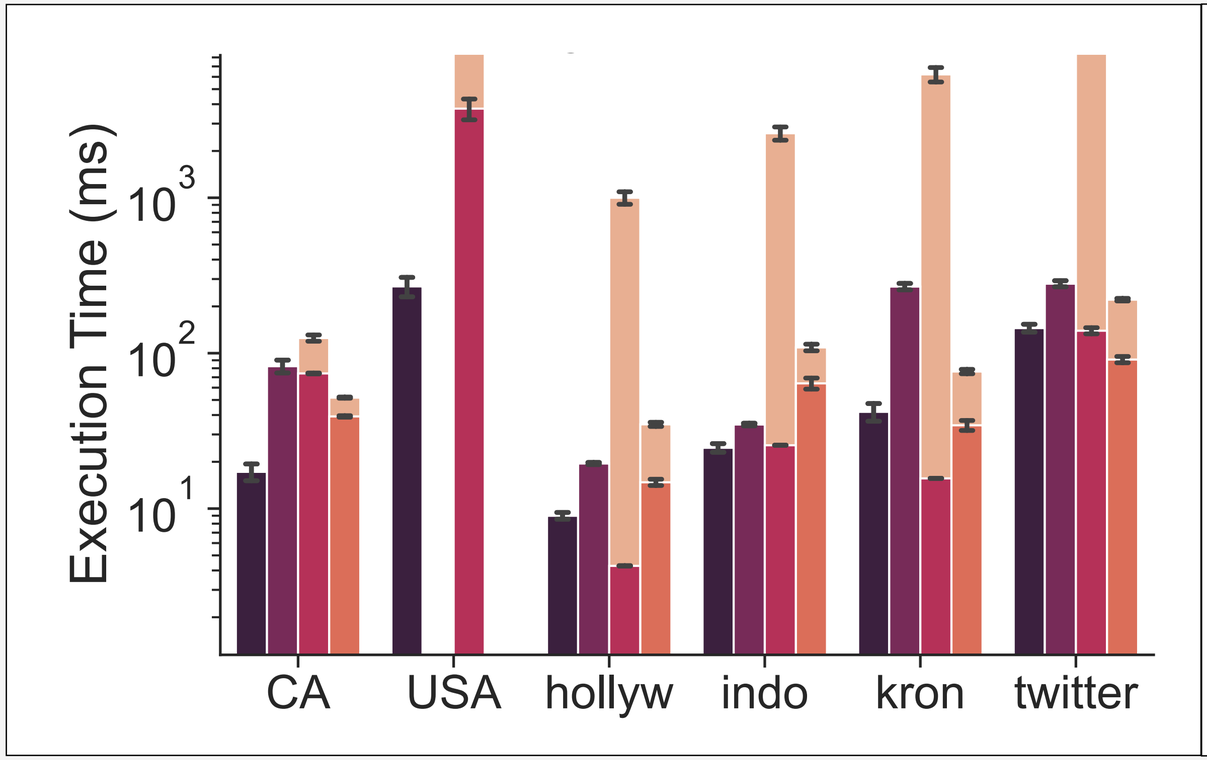


Single Source Shortest Path (SSSP)

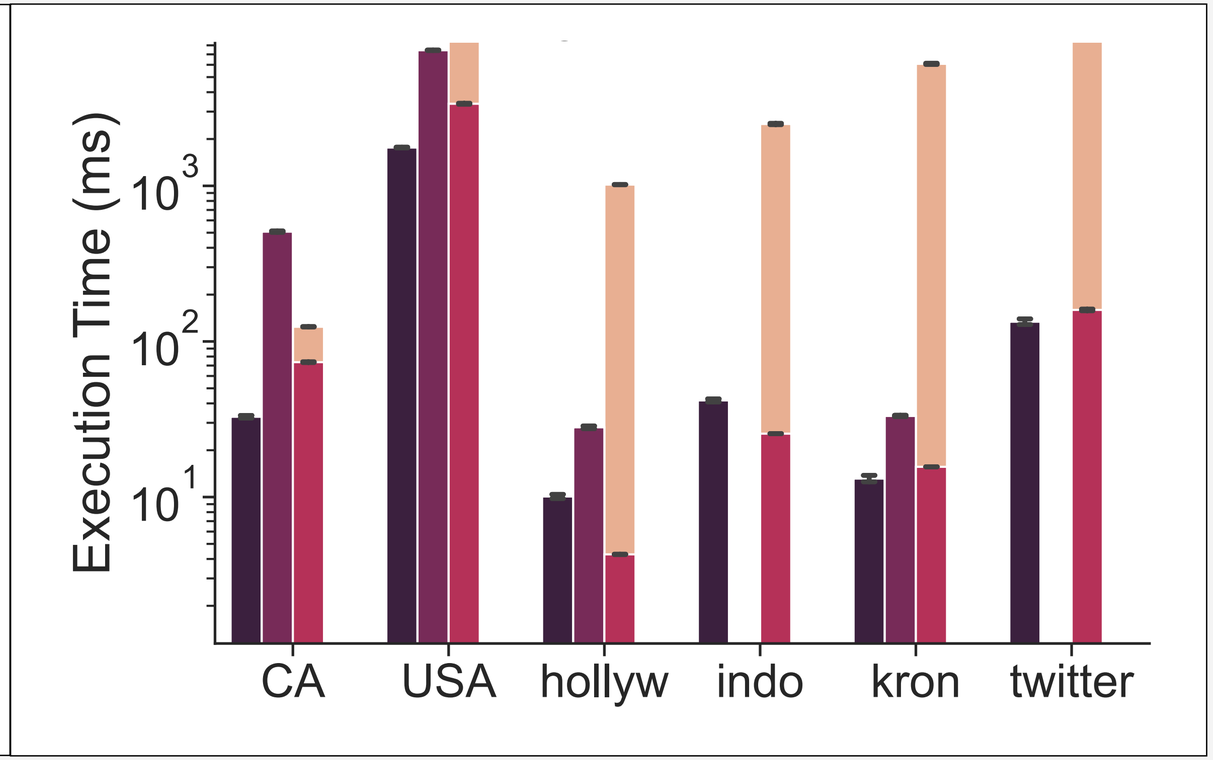


Experimental Evaluation — Performance vs. SOTA Frameworks

Betweenness Centrality (BC)

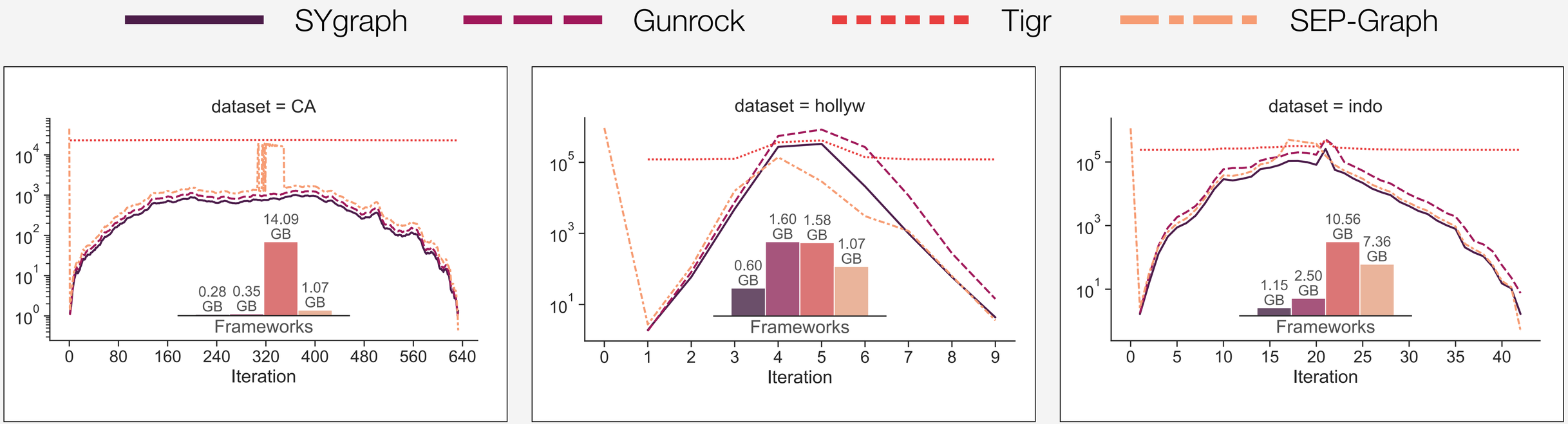


Connected Components Labelling (CC)



Experimental Evaluation — Memory Footprint vs. SOTA Frameworks

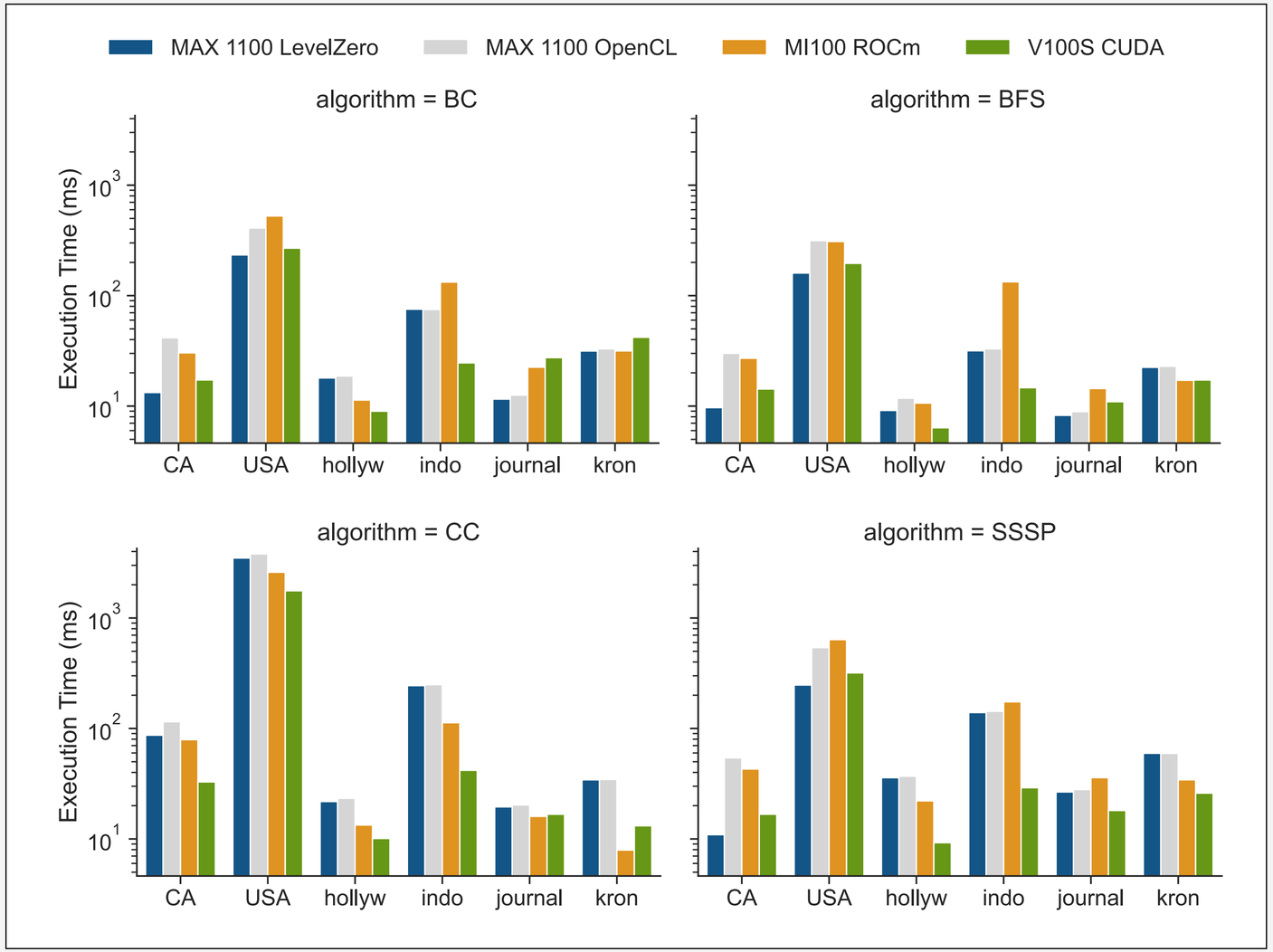
- Memory Footprint expressed in Kilobytes (KB).



Note: Insets bar-plots show total memory usage per framework, following legend order.

Experimental Evaluation — Comparison on Different Hardware

- **Intel MAX 1100** performs well on *sparse workloads* and *sparse graphs* with LevelZero backend
- LevelZero backend yields superior performance compared to OpenCL backend.
- **AMD MI100** excels on *dense workloads*
- **NVIDA V100S** shows overall strong performance



Conclusion

Conclusion

- **SYgraph Summary**
  - A portable graph analytics framework
  - Introduces a Two-Layer Bitmap frontier layout
  - A load balancing mechanism tailored on top of the bitmap
- **Key Results**
  - Up to $3.5\times$ faster than Gunrock, $7.5\times$ over Tigr, $2.3\times$ over SEP-Graph
  - Low memory footprint, no need for preprocessing
  - Demonstrates performance portability

- **Future Work**
  - Support for multi-GPU and multi-node execution
  - Exploring asynchronous and dynamic graph capabilities
  - Auto-tuning mechanism across architectures
  - Integration to oneAPI Data Parallel Library (oneDAL)