34th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing
Babeș-Bolyai University • Cluj-Napoca • Romania • 26 March 2026

Evaluating Portable Programming Models for Hypergraph Label Propagation on GPUs

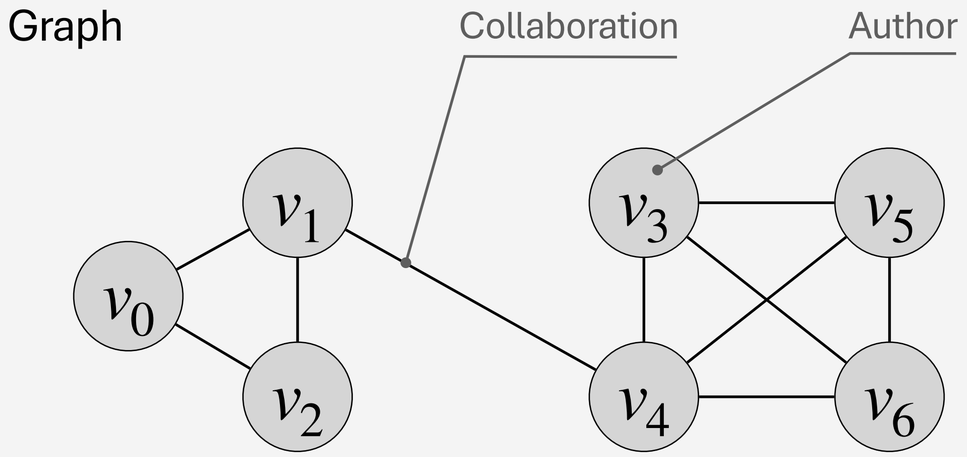
Antonio De Caro[1]***,*** *Dario De Maio*[1]*, Francesco Monzillo*[1]*, Alessia Antelmi*[2]*, Biagio Cosenza*[1]

1 University of Salerno • Fisciano • Italy

2 University of Turin Turin • Italy

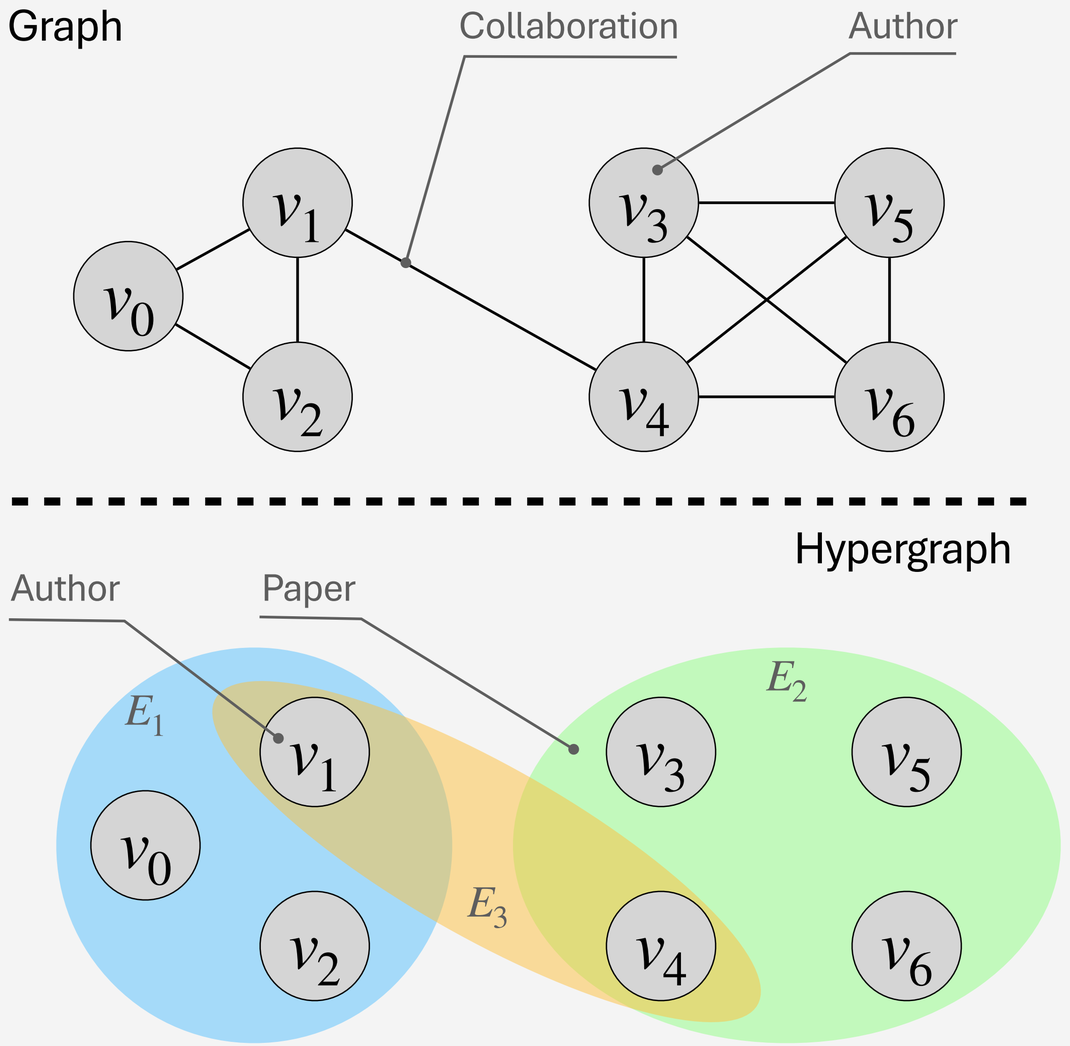
Why Hypergraphs?

- Graphs only capture pairwise interactions



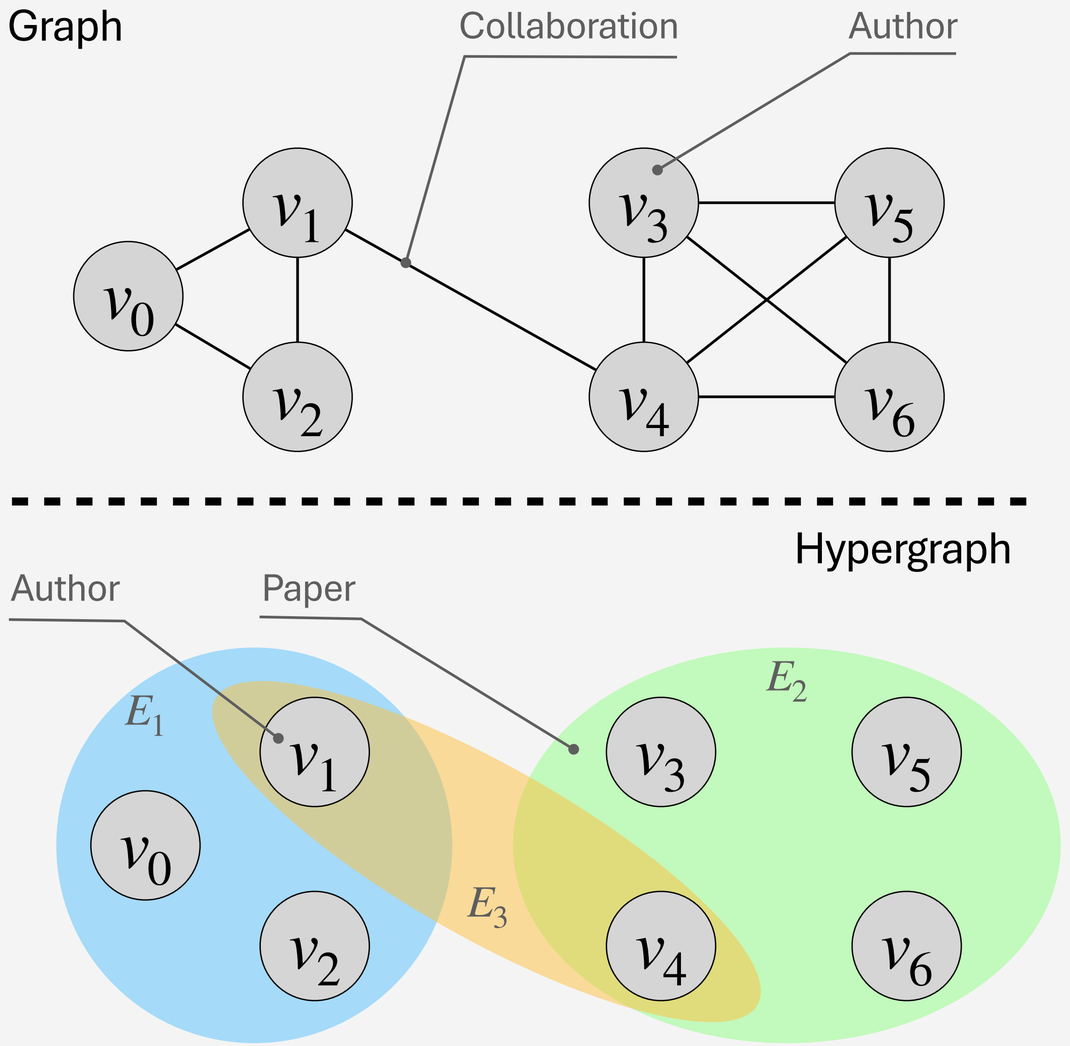
Why Hypergraphs?

- Graphs only capture pairwise interactions
- Many real systems benefit from higher-order relationships
  → hypergraphs



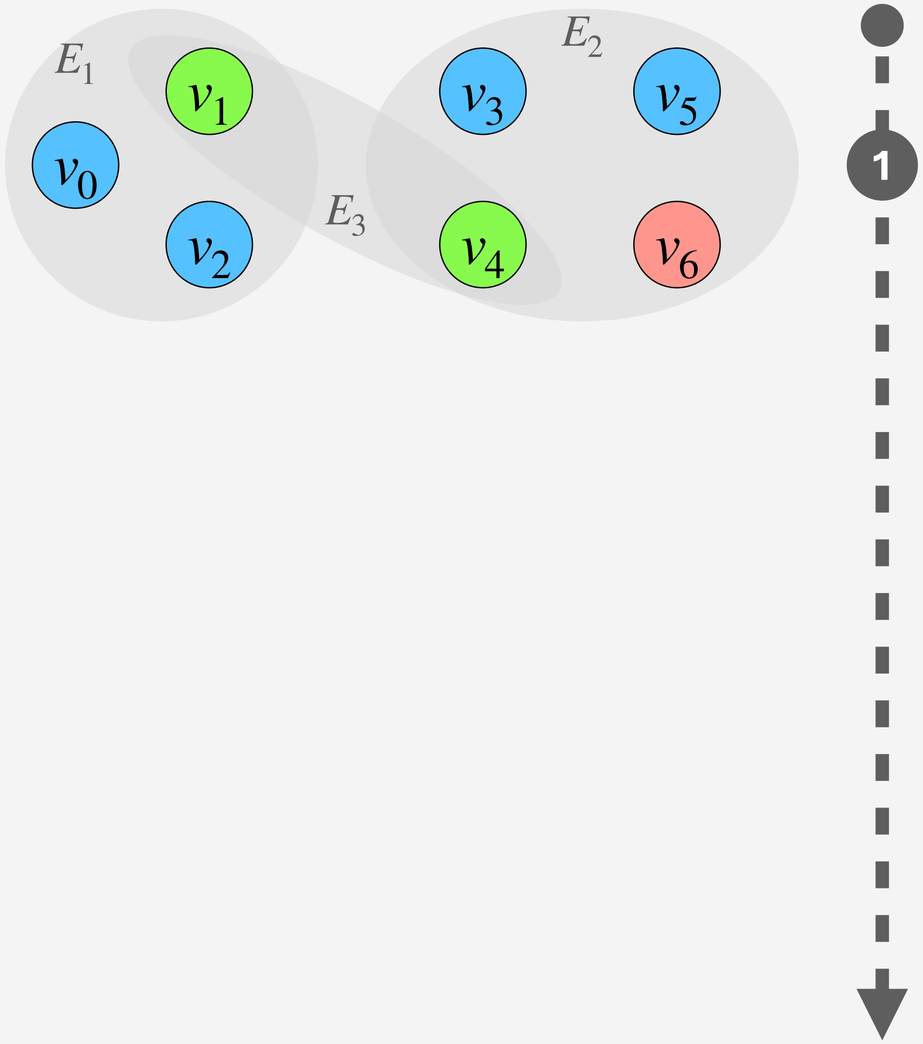
Why Hypergraphs?

- Graphs only capture pairwise interactions
- Many real systems benefit from higher-order relationships
  → hypergraphs
- Community detection groups nodes that interact more with each other
  - Functional roles
  - Hidden structure
  - Patterns of Collaboration or Influence



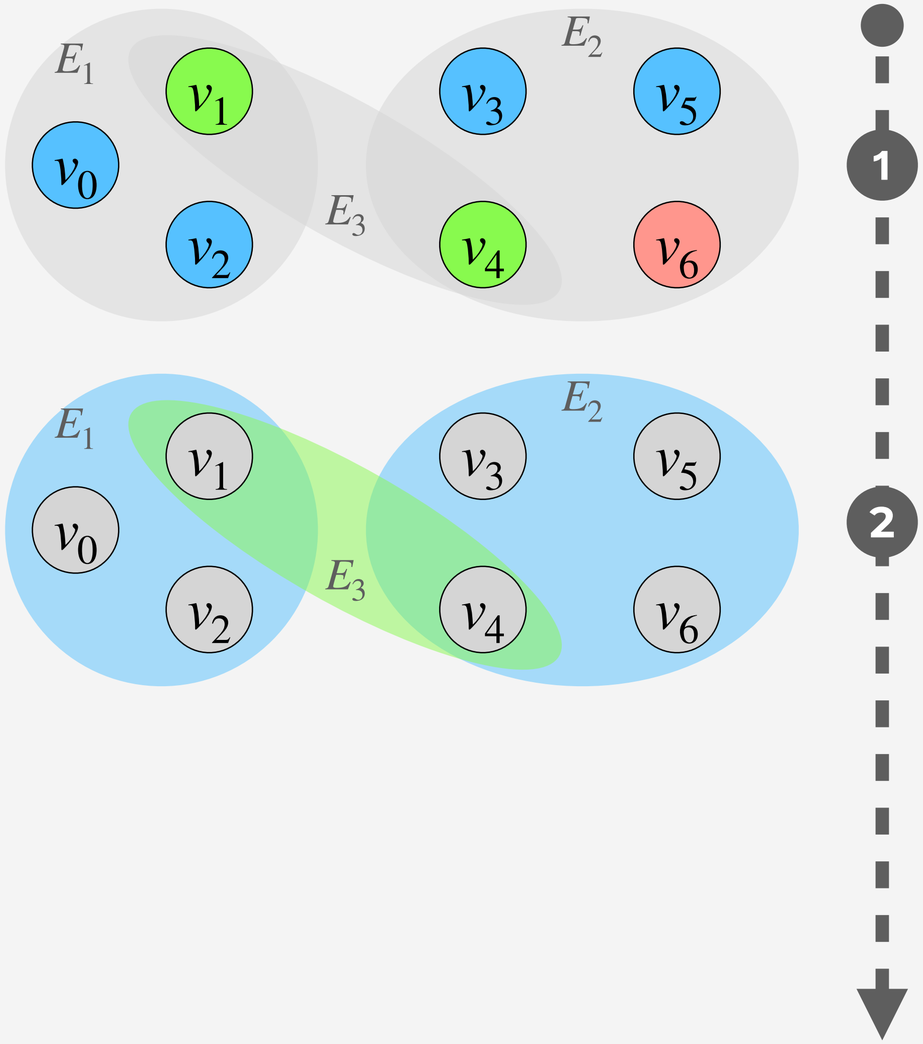
Community Detection via Label Propagation (HLP)

1. Initialize vertex labels



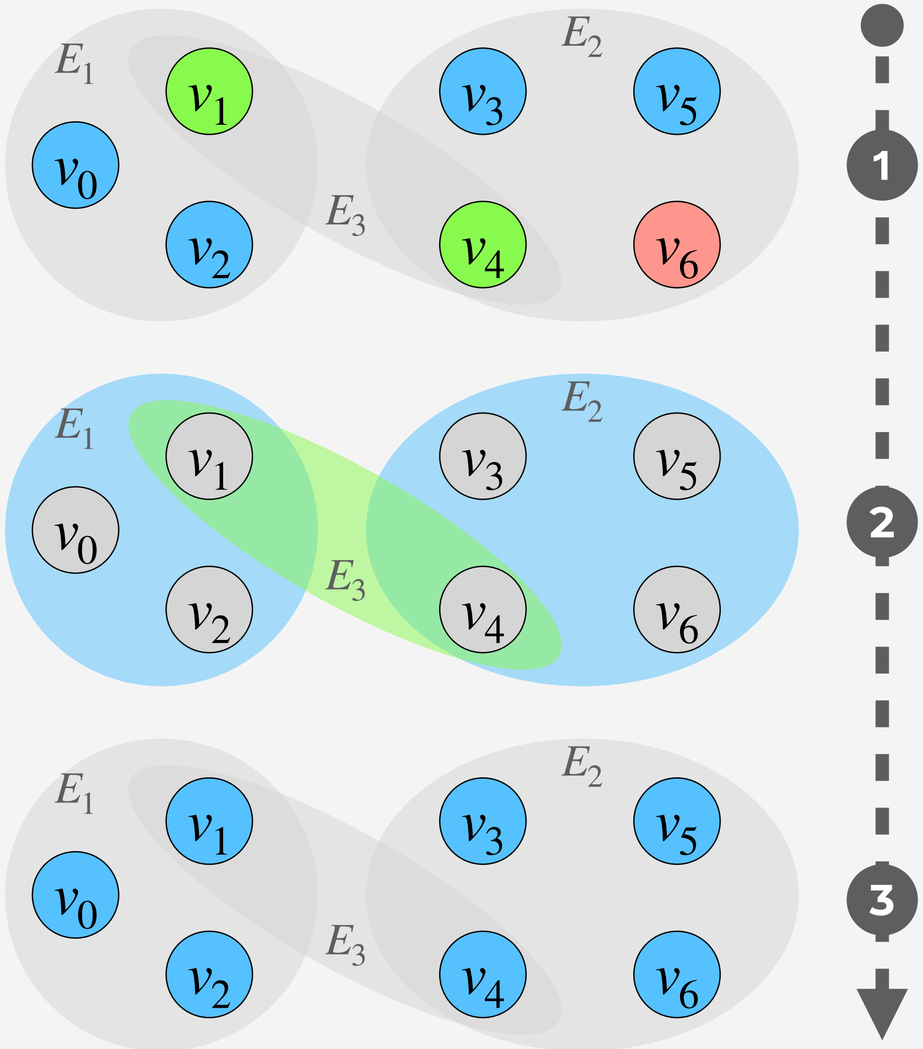
Community Detection via Label Propagation (HLP)

1. Initialize vertex labels
2. **Edge phase**
   - each hyperedge selects **most frequent label** among its vertices



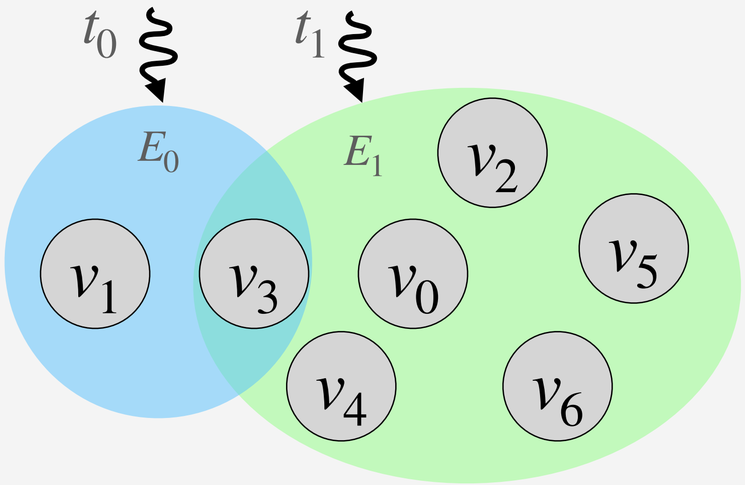
Community Detection via Label Propagation (HLP)

1. Initialize vertex labels
2. **Edge phase**
   - each hyperedge selects **most frequent label** among its vertices
3. **Vertex phase**
   - each vertex selects the **most frequent label** among its hyperedges
4. Repeat until **convergence** or a until the number of changed labels falls below a given **threshold**



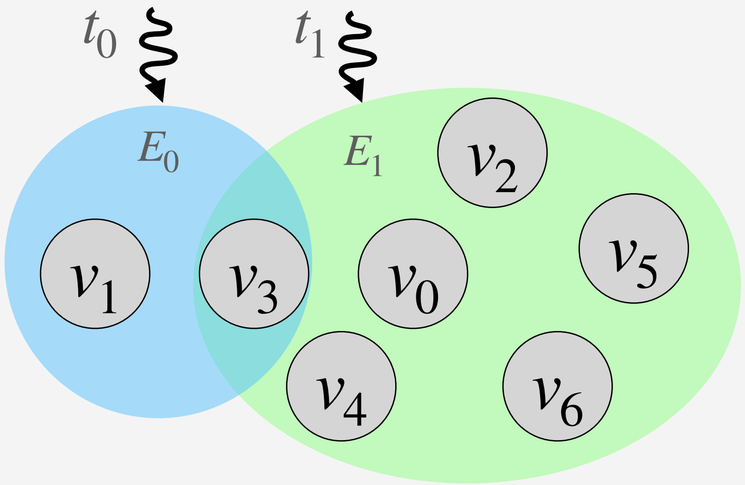
Challenges of Hypergraphs on GPUs

- GPUs are attractive for scale, but hypergraph workloads are
  - Sparse
  - Irregular
  - Memory Bound



Challenges of Hypergraphs on GPUs

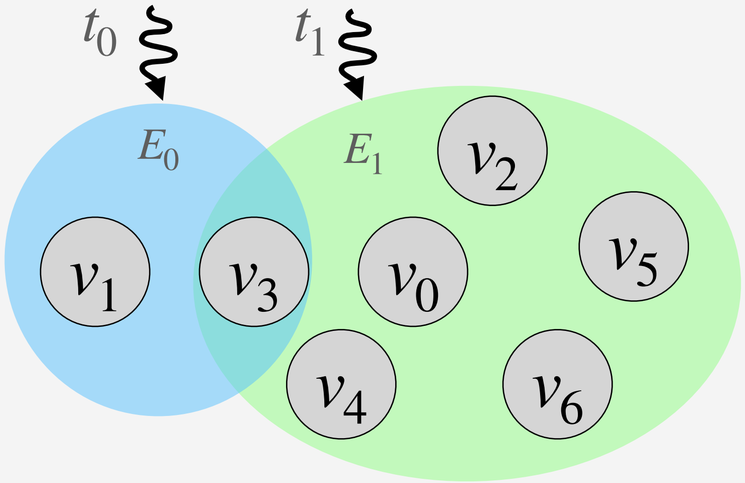
- GPUs are attractive for scale, but hypergraph workloads are
  - Sparse
  - Irregular
  - Memory Bound



- Modern systems include:
  - AMD GPUs *(ElCapitan #1 Top500)*
  - Intel GPUs *(Aurora #2 Top500)*
  - NVIDIA GPUs *(JUPITER #4 Top500)*

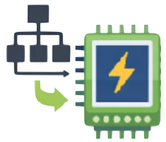
Challenges of Hypergraphs on GPUs

- GPUs are attractive for scale, but hypergraph workloads are
  - Sparse
  - Irregular
  - Memory Bound



- Modern systems include:
  - AMD GPUs *(ElCapitan #1 Top500)*
  - Intel GPUs *(Aurora #2 Top500)*
  - NVIDIA GPUs *(JUPITER #4 Top500)*

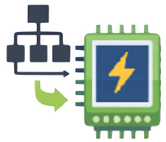
This Work Contributions



HLP GPU Optimizations

- CSR + CSC
- Degree Decomp.
- Shared Memory

This Work Contributions



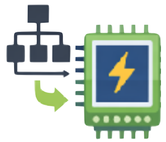
HLP GPU Optimizations

- CSR + CSC
- Degree Decomp.
- Shared Memory

Portable Implementations

- SYCL
- OpenMP
- Kokkos

This Work Contributions



HLP GPU Optimizations

- CSR + CSC
- Degree Decomp.
- Shared Memory

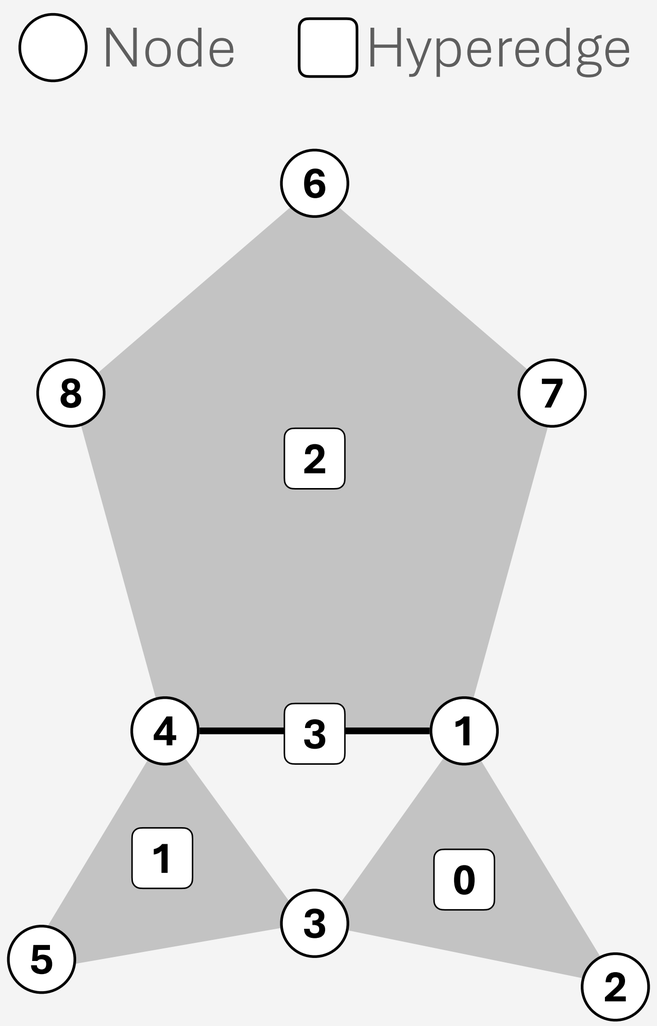
Portable Implementations

- SYCL
- OpenMP
- Kokkos

Experimental Evaluation

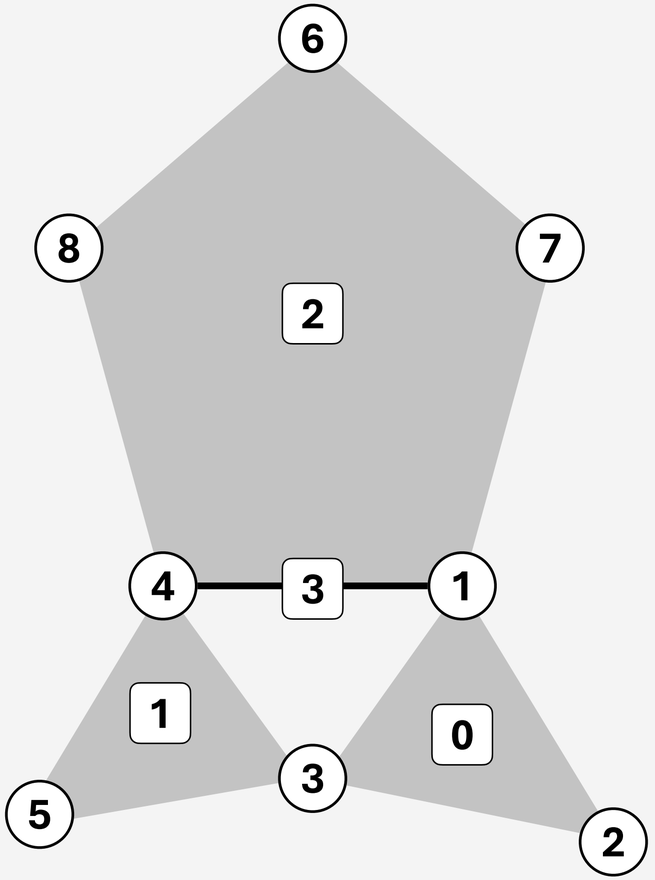
- Performance
- Portability
- Productivity

Optimization #1 — *Hypergraph Representation*



Optimization #1 — *Hypergraph Representation*

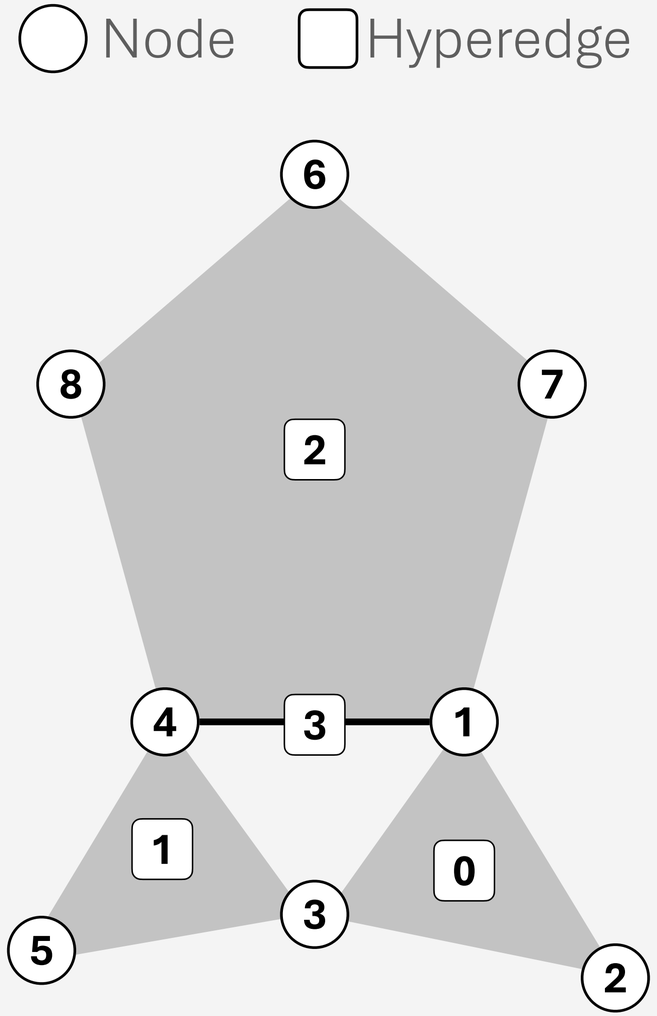
Node Hyperedge



Hyperedges

Nodes	0	1	2	3
1	1		1	1
2	1			
3	1	1		
4		1	1	1
5		1		
6			1	
7			1	
8			1	

Optimization #1 — *Hypergraph Representation*



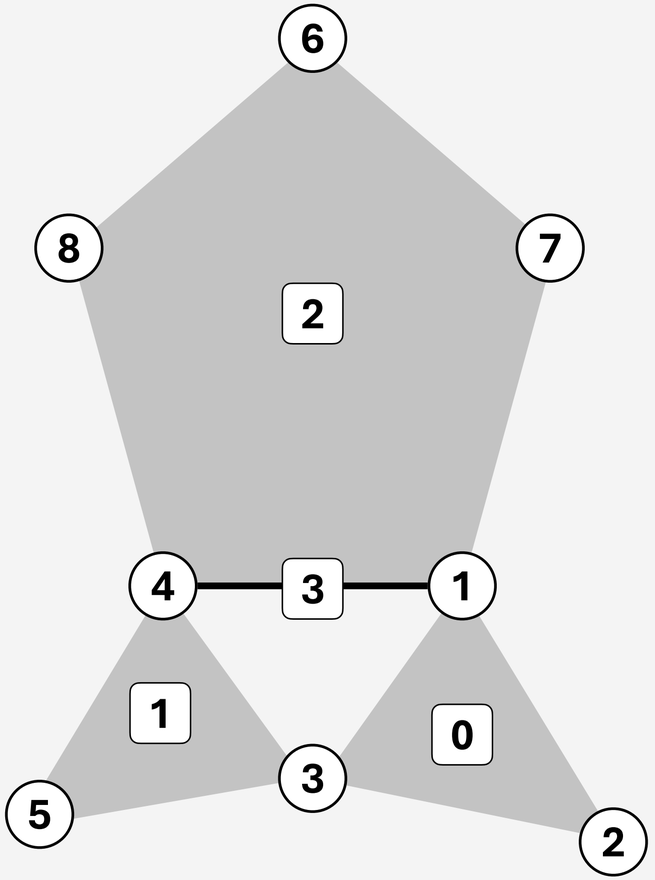
Too many non-zeros

Hyperedges

Nodes	0	1	2	3
1	1		1	1
2	1			
3	1	1		
4		1	1	1
5		1		
6			1	
7			1	
8			1	

Optimization #1 — *Hypergraph Representation*

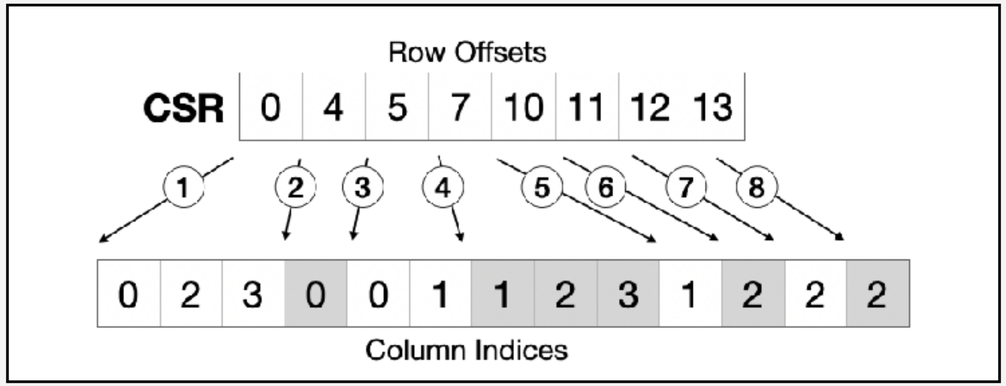
Node Hyperedge



Too many non-zeros

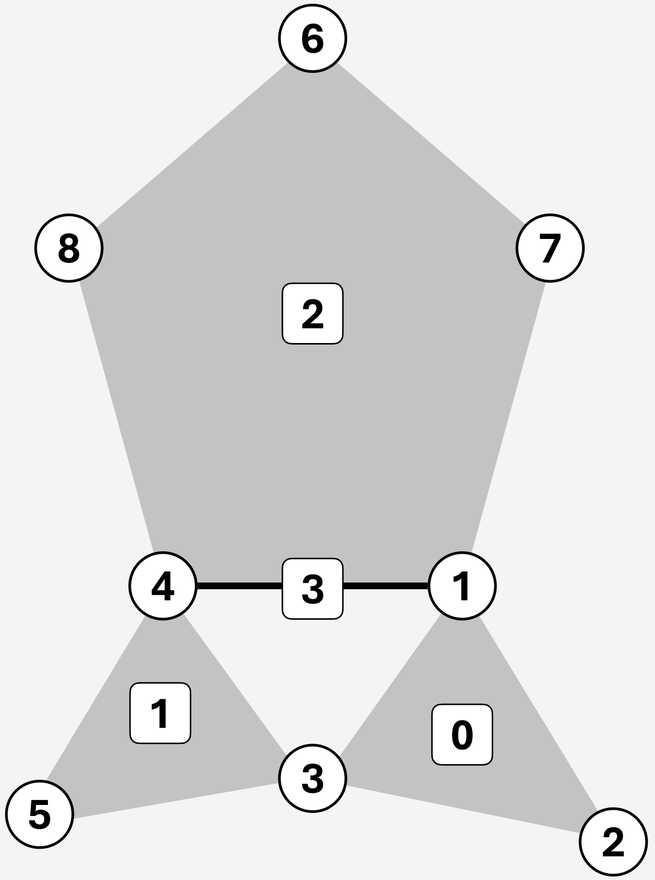
Hyperedges

Nodes	0	1	2	3
1	1		1	1
2	1			
3	1	1		
4		1	1	1
5		1		
6			1	
7			1	
8			1	



Optimization #1 — *Hypergraph Representation*

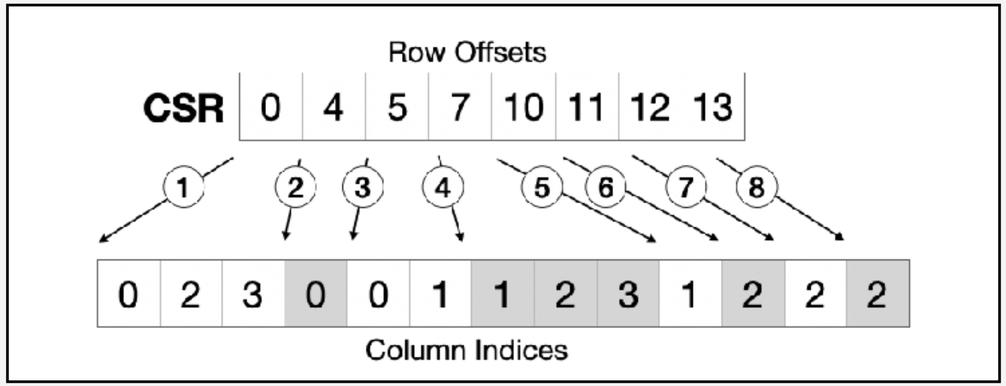
Node Hyperedge



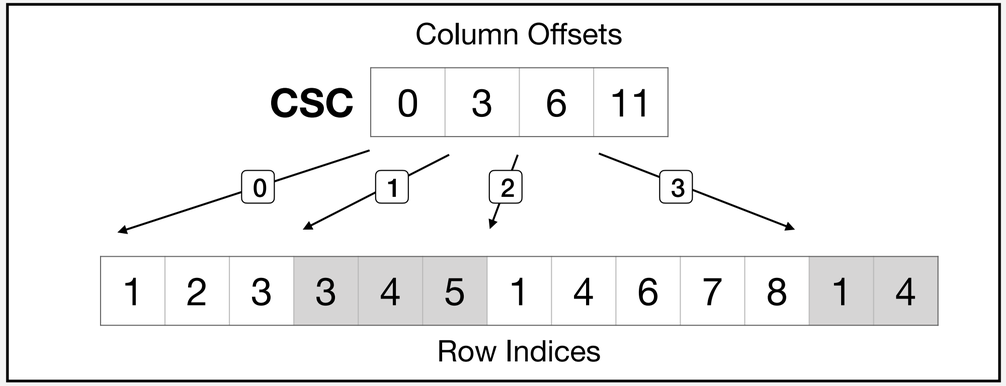
Too many non-zeros

Hyperedges

Nodes	0	1	2	3
1	1		1	1
2	1			
3	1	1		
4		1	1	1
5		1		
6			1	
7			1	
8			1	

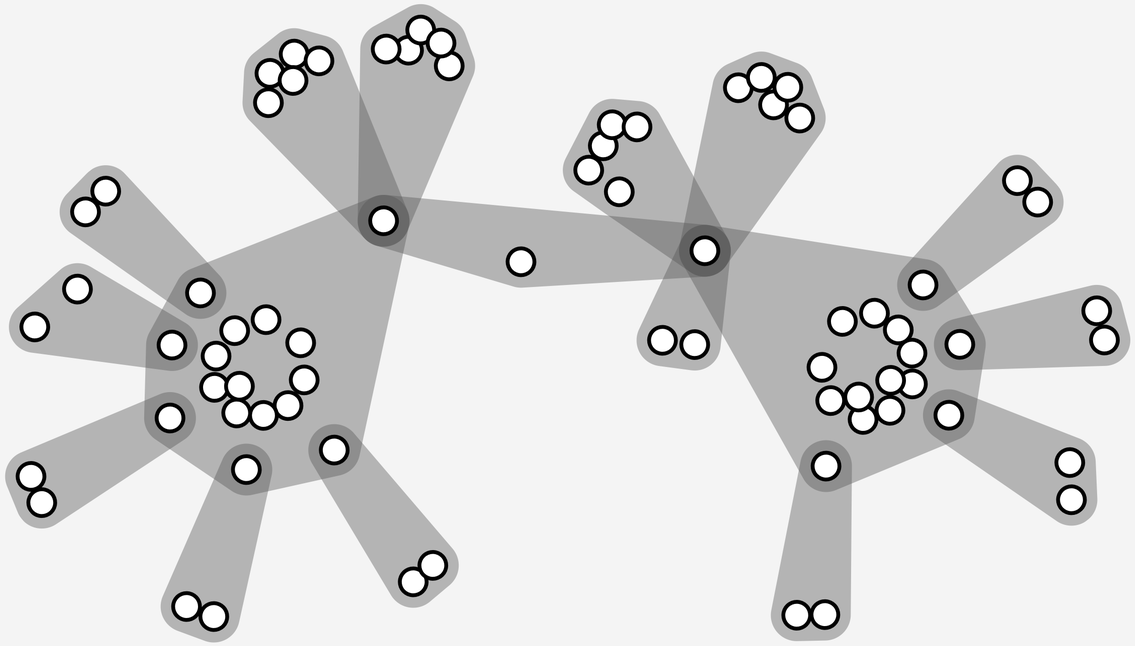


+



Optimization #2 — *Phase Decomposition*

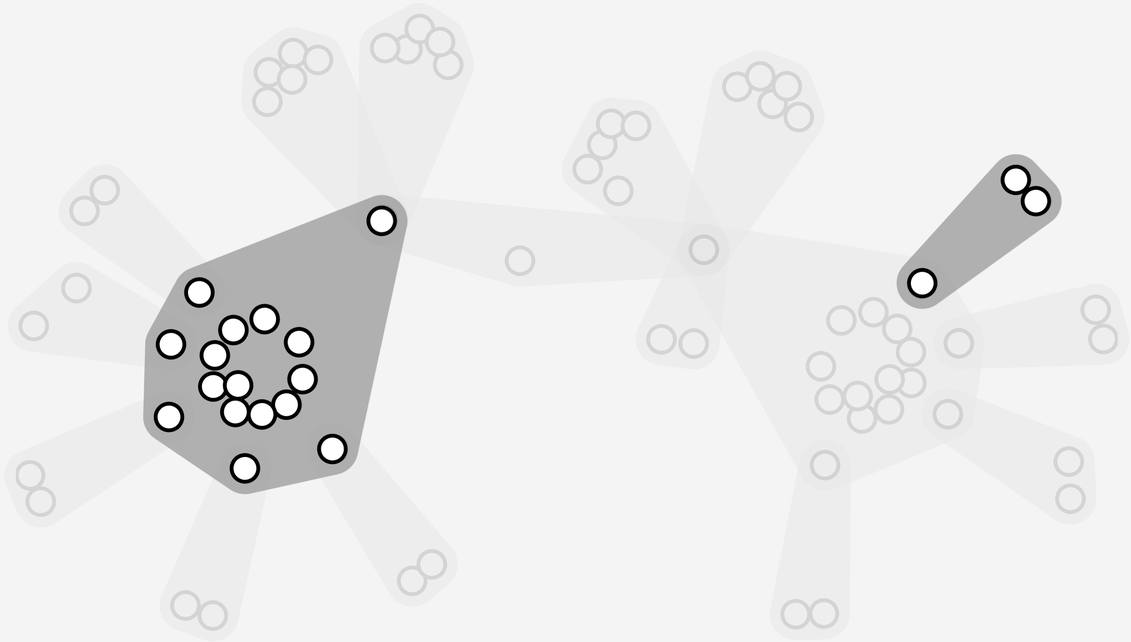
- Assigning threads to a different vertex/hyperedge



Optimization #2 — *Phase Decomposition*

- Assigning threads to a different vertex/hyperedge

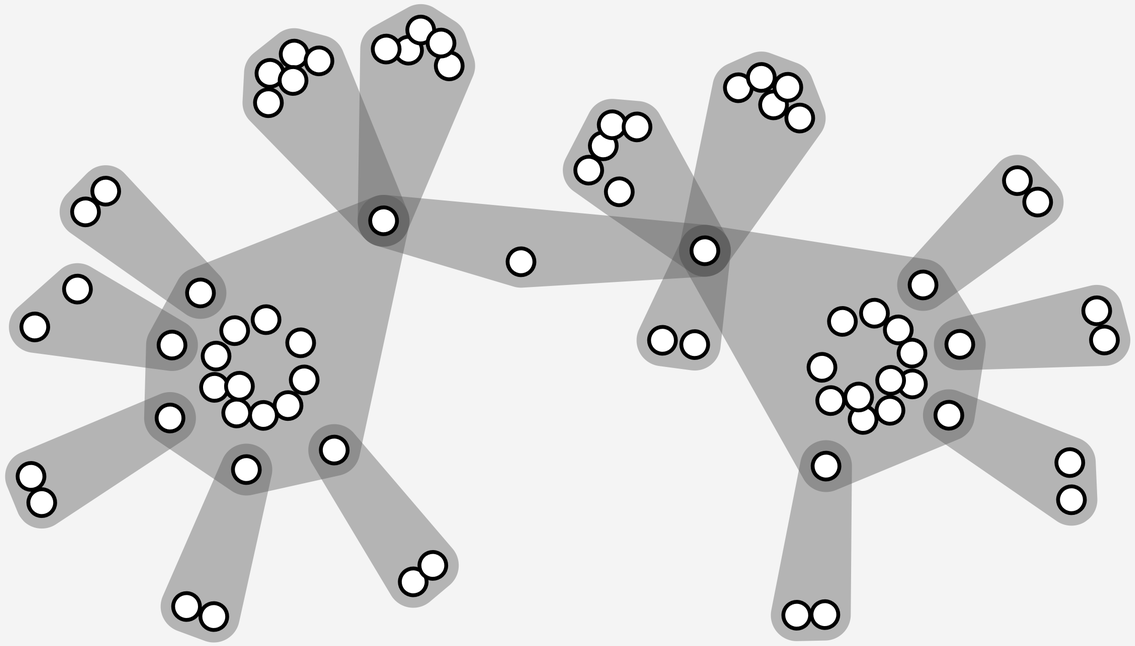
➡**Severe Load Imbalance**



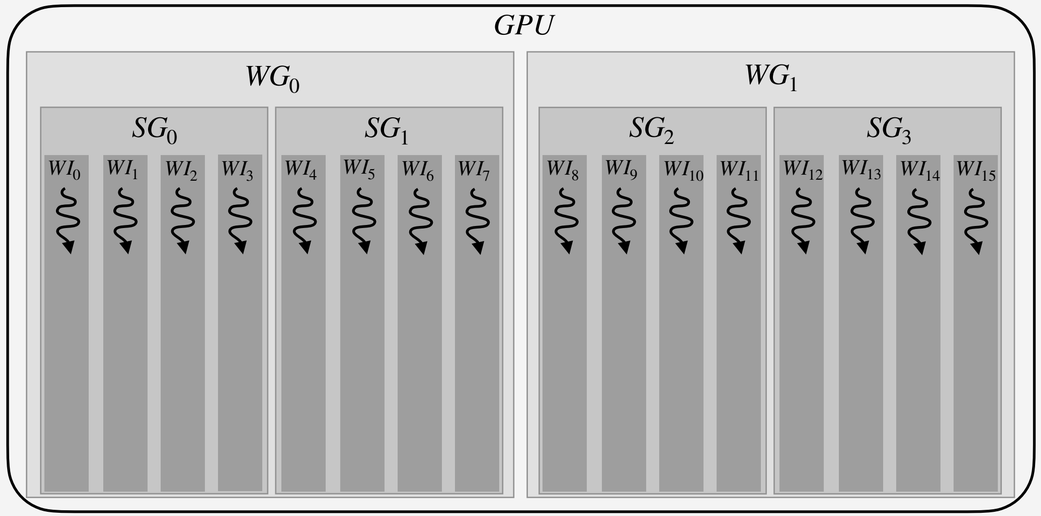
Optimization #2 — *Phase Decomposition*

- Assigning threads to a different vertex/hyperedge

➡**Severe Load Imbalance**



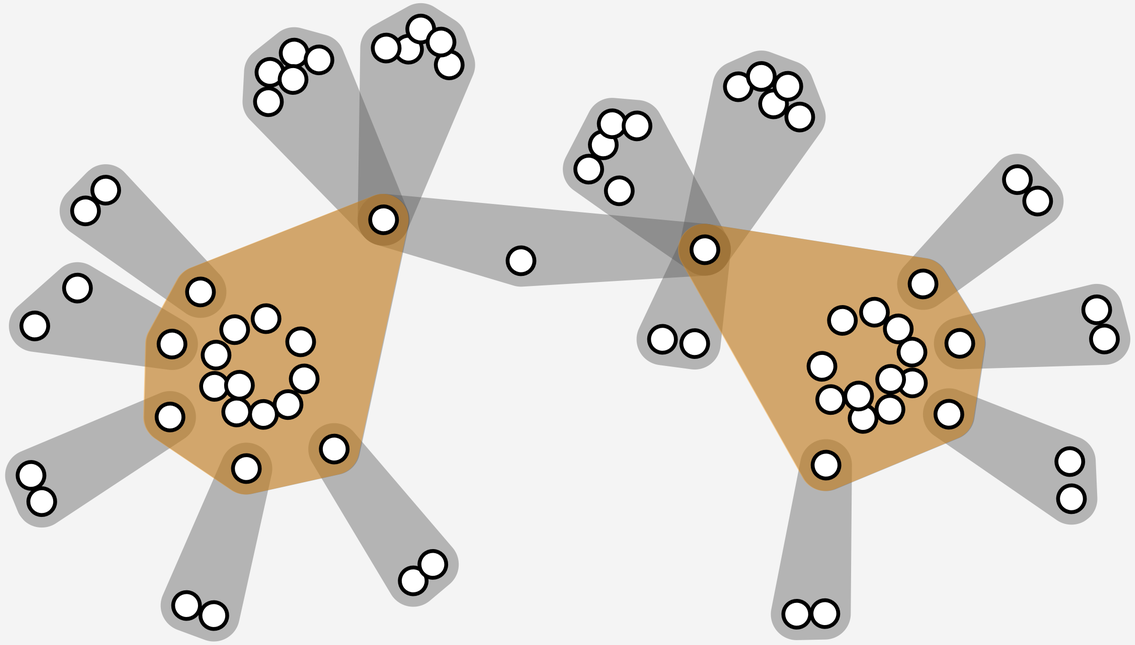
- Computation divided in **3 Kernels**:



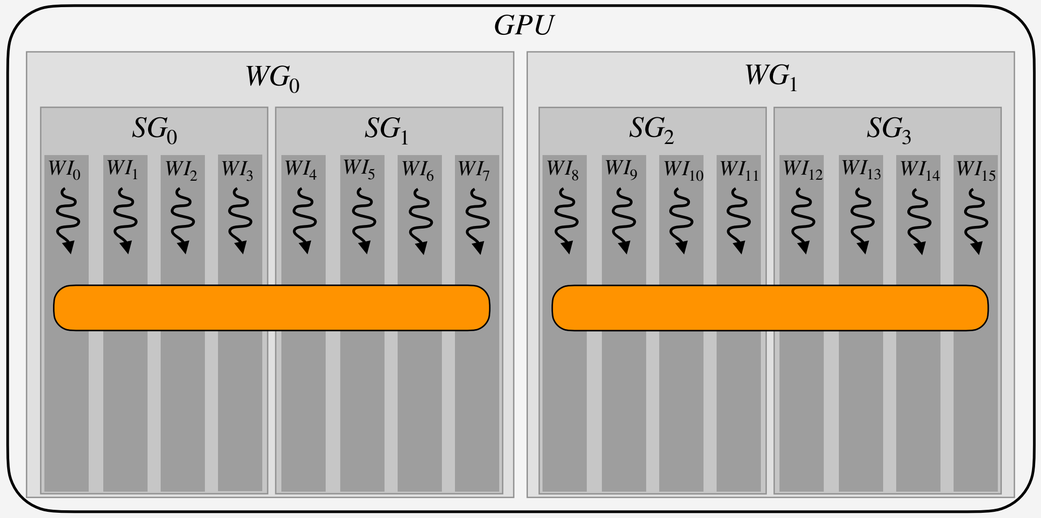
Optimization #2 — *Phase Decomposition*

- Assigning threads to a different vertex/hyperedge

  ➡**Severe Load Imbalance**



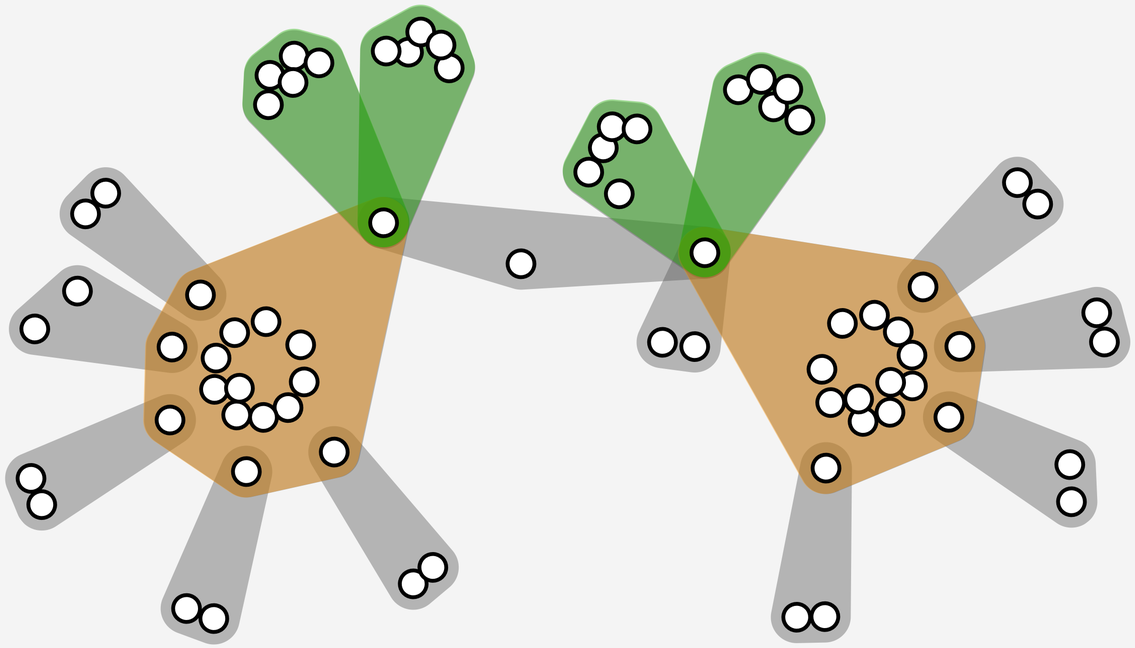
- Computation divided in **3 Kernels**:
  - *Large-degree*: one work-group (block)



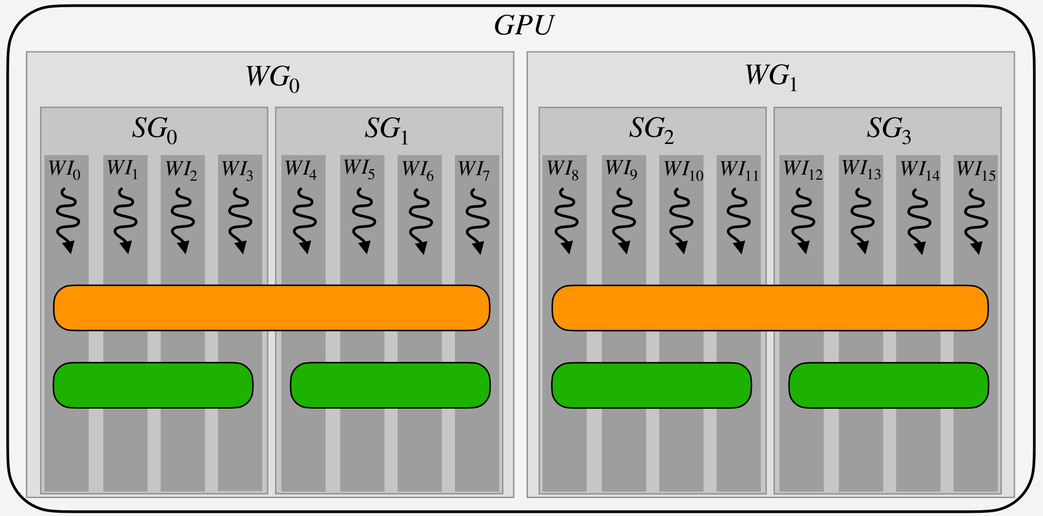
Optimization #2 — *Phase Decomposition*

- Assigning threads to a different vertex/hyperedge

  ➡**Severe Load Imbalance**



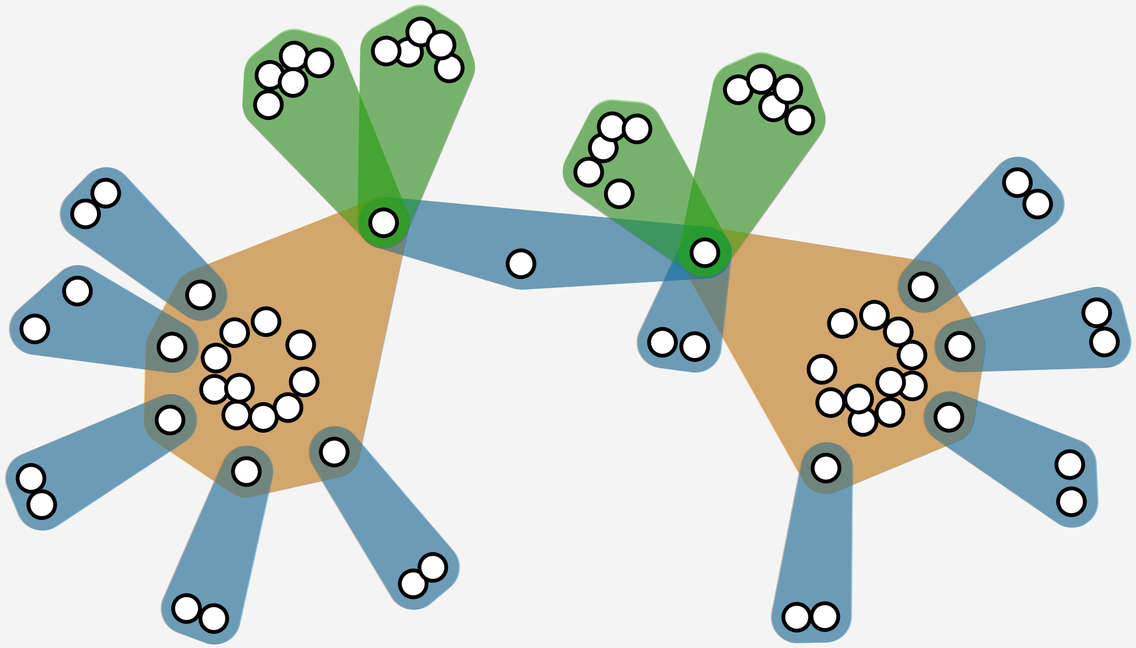
- Computation divided in **3 Kernels**:
  - *Large-degree*: one work-group (block)
  - *Medium-degree*: one sub-group (warp)



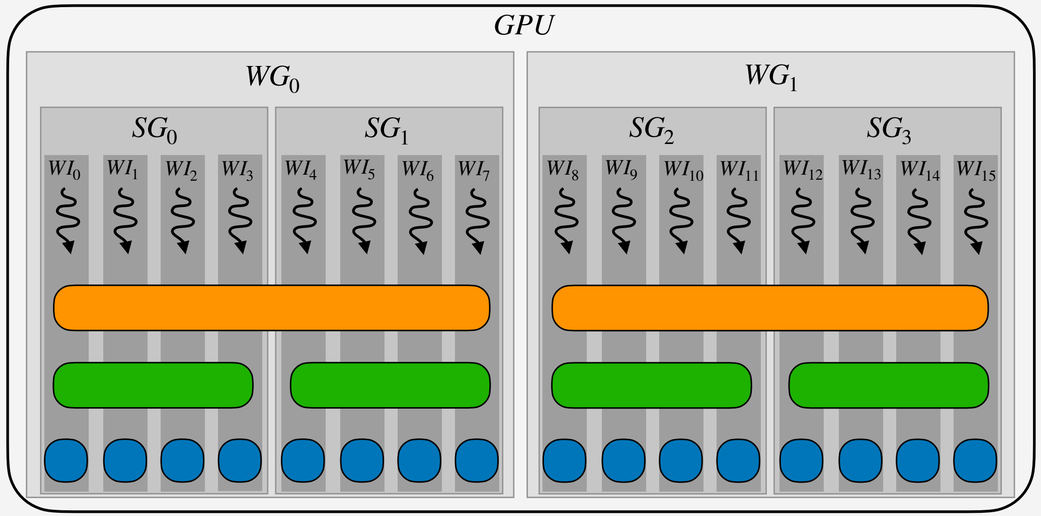
Optimization #2 — *Phase Decomposition*

- Assigning threads to a different vertex/hyperedge

  ➡**Severe Load Imbalance**

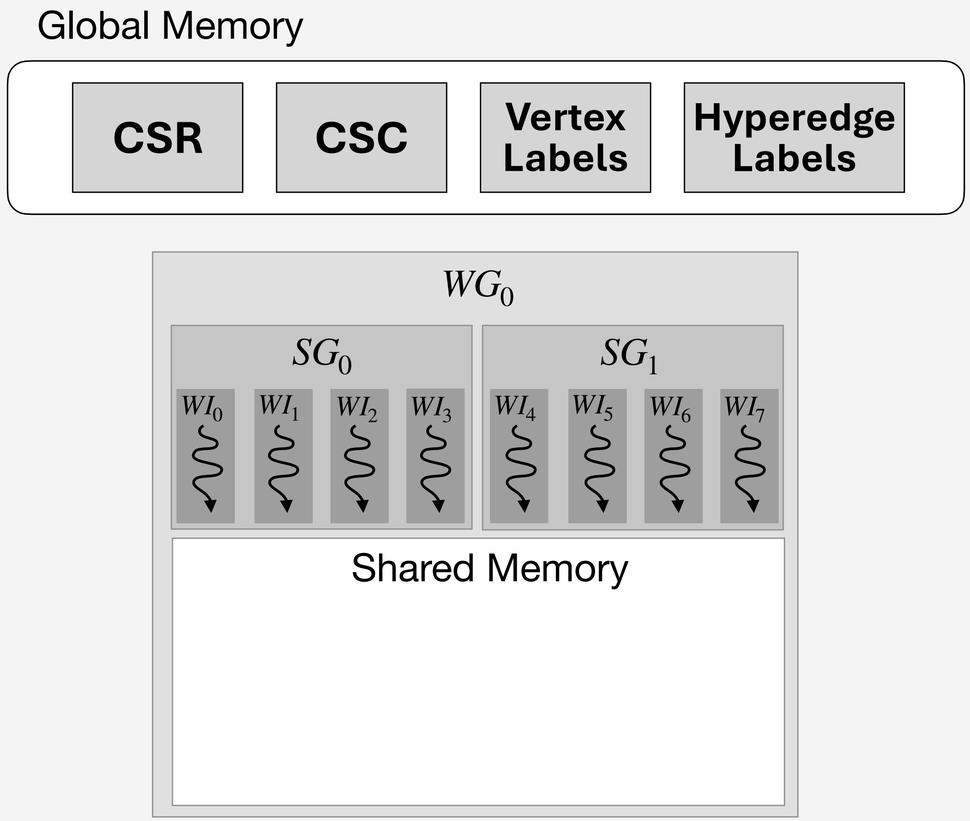


- Computation divided in **3 Kernels**:
  - *Large-degree*: one work-group (block)
  - *Medium-degree*: one sub-group (warp)
  - *Small-degree*: one work-item (thread)



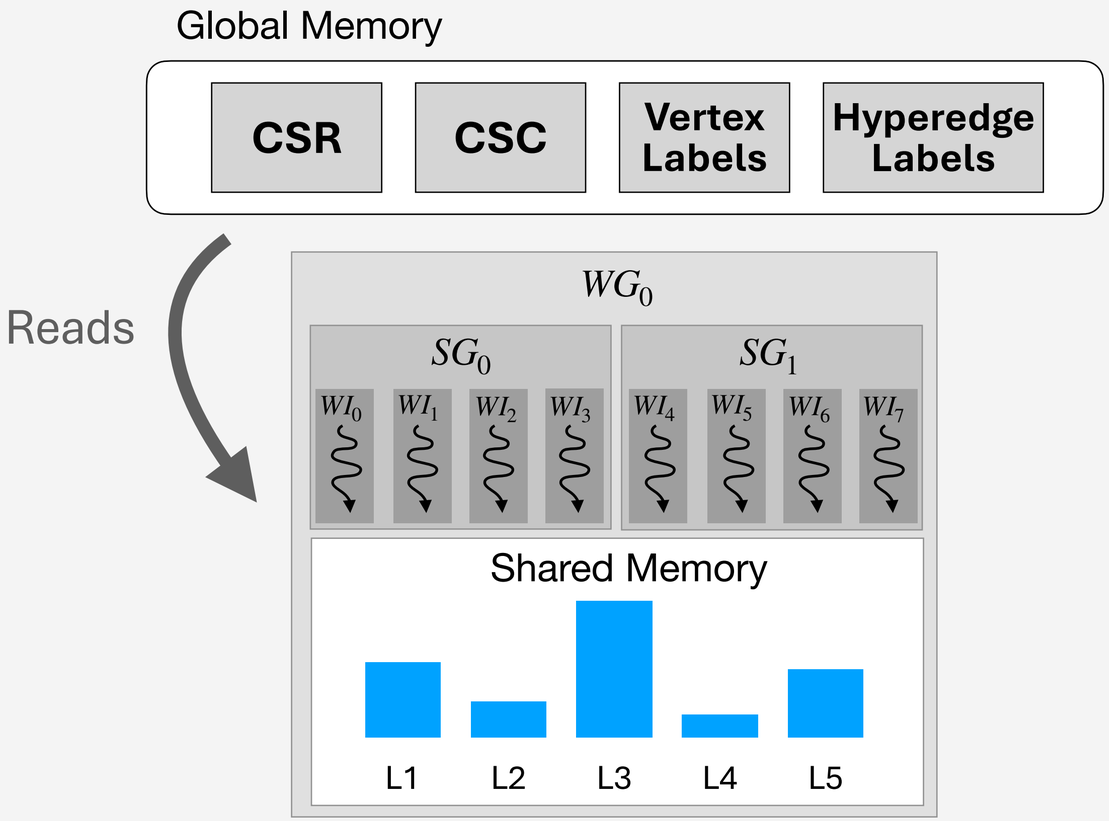
Optimization #3 — *Label Aggregation*

- Label Aggregation requires **frequent memory accesses**
  - Global Memory is slow



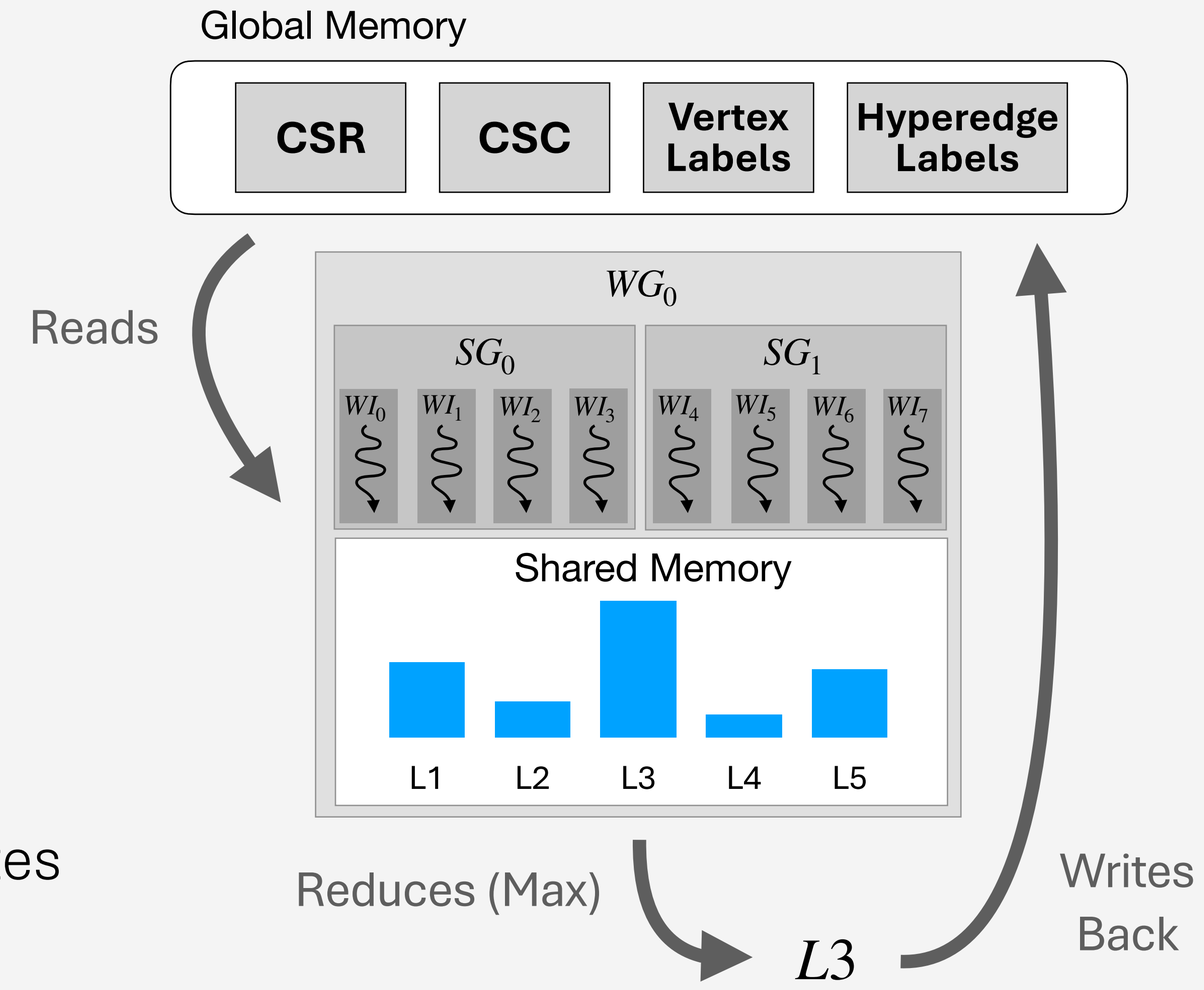
Optimization #3 — *Label Aggregation*

- Label Aggregation requires **frequent memory accesses**
  - Global Memory is slow
- Use **Shared (Local) Memory**
  - Stores temporary label histograms
  - Reduce global memory traffic
  - Reuse data within thread groups



Optimization #3 — *Label Aggregation*

- Label Aggregation requires **frequent memory accesses**
  - Global Memory is slow
- Use **Shared (Local) Memory**
  - Stores temporary label histograms
  - Reduce global memory traffic
  - Reuse data within thread groups
- **Reduces** the most frequent label and writes back to global memory



Implementation across Programming Models

SYCL	Kokkos	OpenMP

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map
Parallelism	Large-degree Bucket	Work-Group	Teams	Teams

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map
Parallelism	Large-degree Bucket	Work-Group	Teams	Teams
	Small-degree Bucket	Work-Item	Threads	Threads

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map
Parallelism	Large-degree Bucket	Work-Group	Teams	Teams
	Small-degree Bucket	Work-Item	Threads	Threads
	Medium-degree Bucket	Sub-Group	Vector Lanes	X

Implementation across Programming Models

		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map
Parallelism	Large-degree Bucket	Work-Group	Teams	Teams
	Small-degree Bucket	Work-Item	Threads	Threads
	Medium-degree Bucket	Sub-Group	Vector Lanes	X
Reductions		`sycl::reduction`	`parallel_reduce`	`reduction(…)` pragma

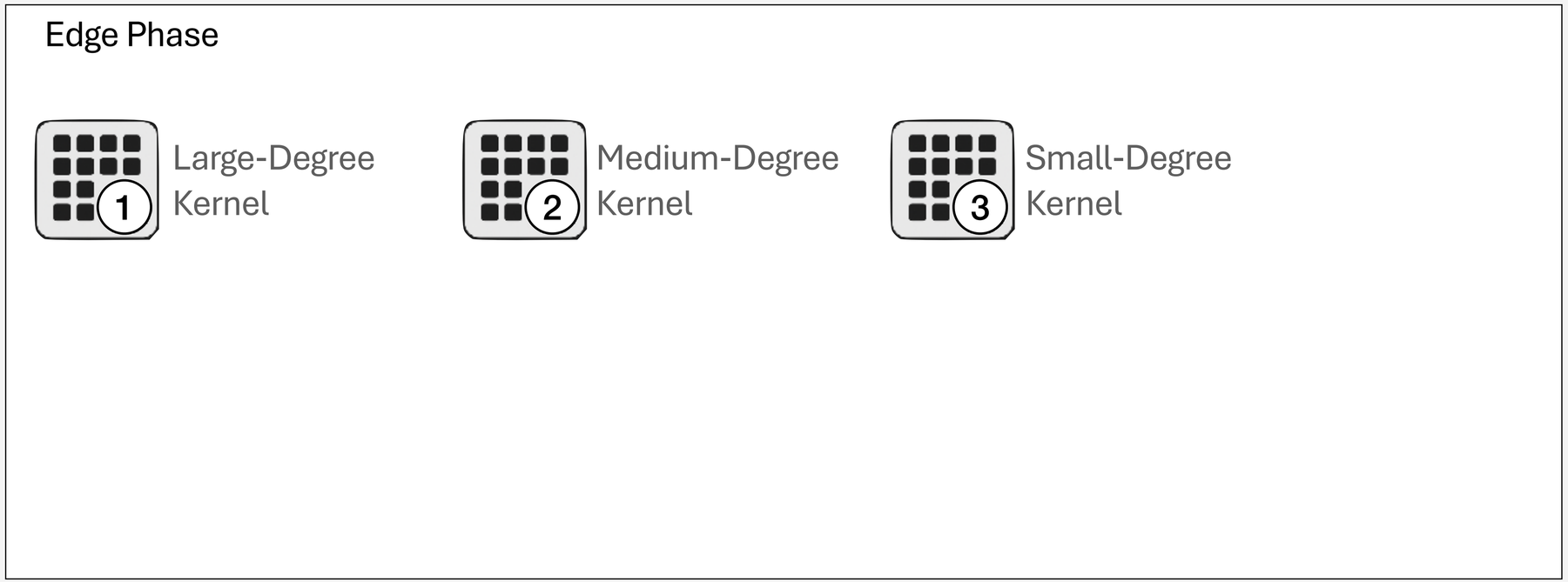
Implementation across Programming Models

208 SLOC | 221 SLOC | 141 SLOC

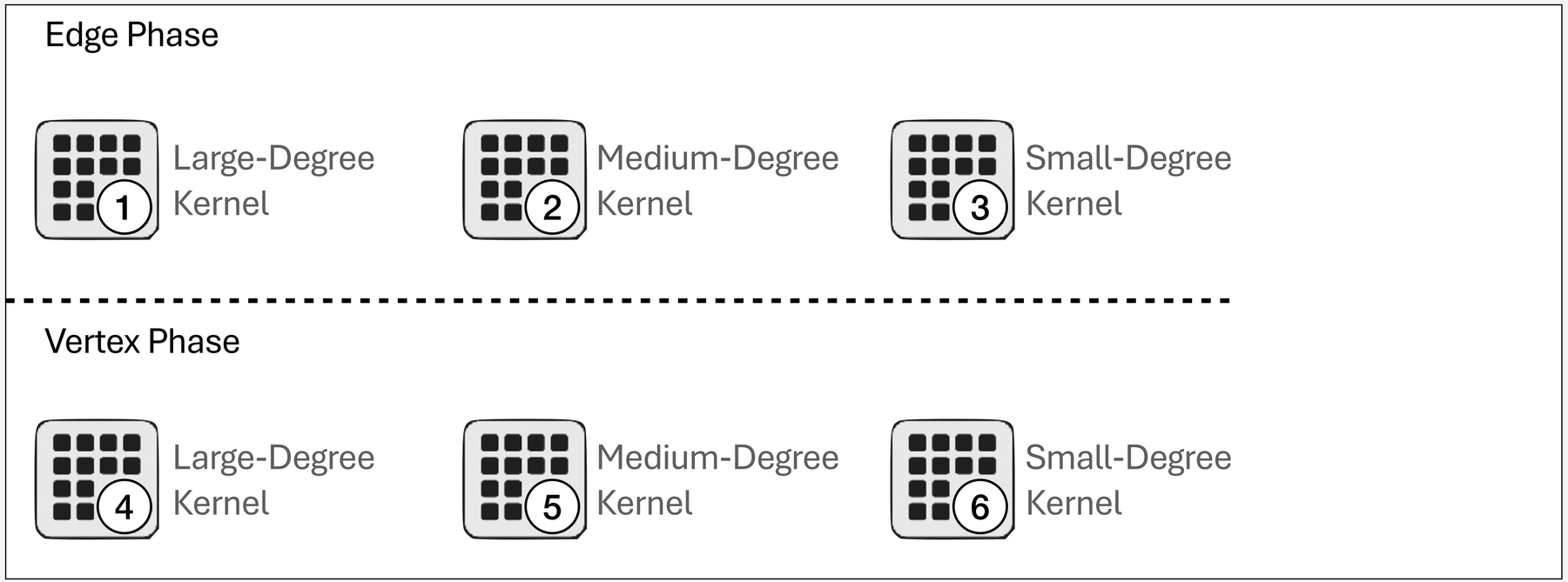
		SYCL	Kokkos	OpenMP
Memory	Global	USM	Views	Map Clauses
	Shared	Local Accessor	Scratch Memory	Shared/Map
Parallelism	Large-degree Bucket	Work-Group	Teams	Teams
	Small-degree Bucket	Work-Item	Threads	Threads
	Medium-degree Bucket	Sub-Group	Vector Lanes	X
Reductions		`sycl::reduction`	`parallel_reduce`	`reduction(…)` `pragma`

Summary Workflow

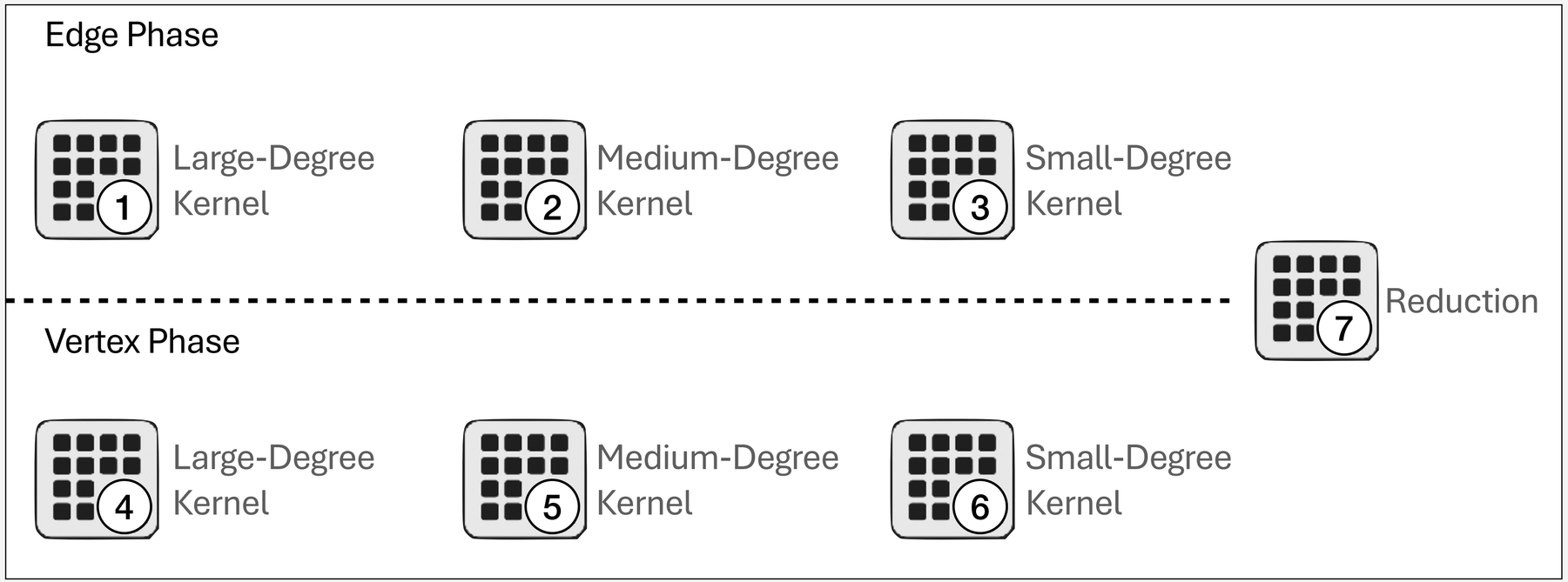
Summary Workflow



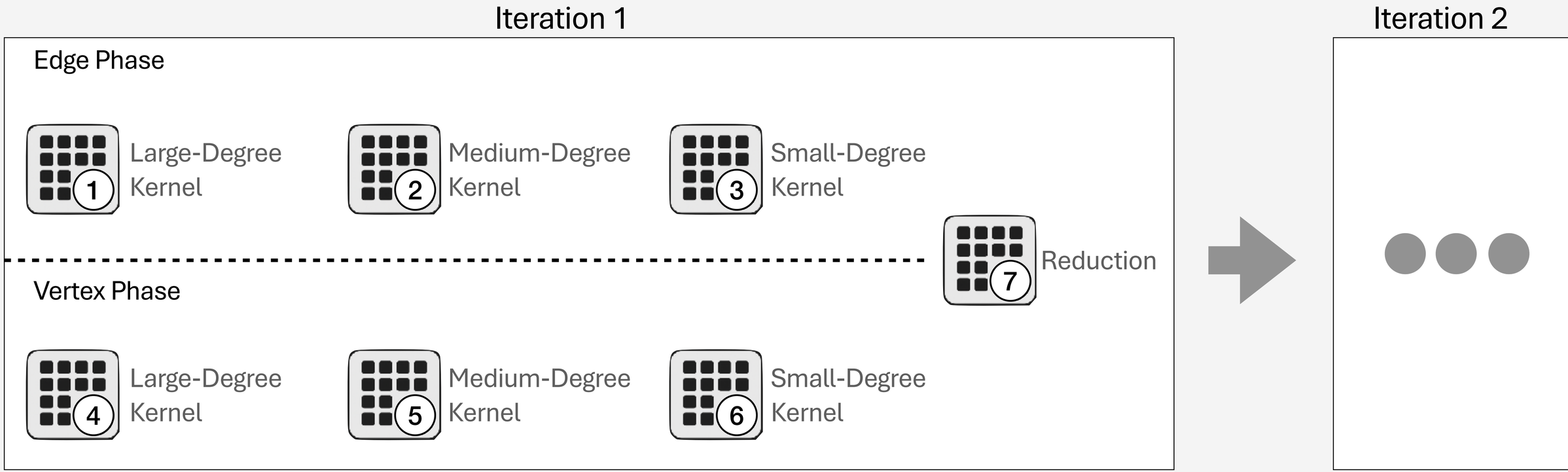
Summary Workflow



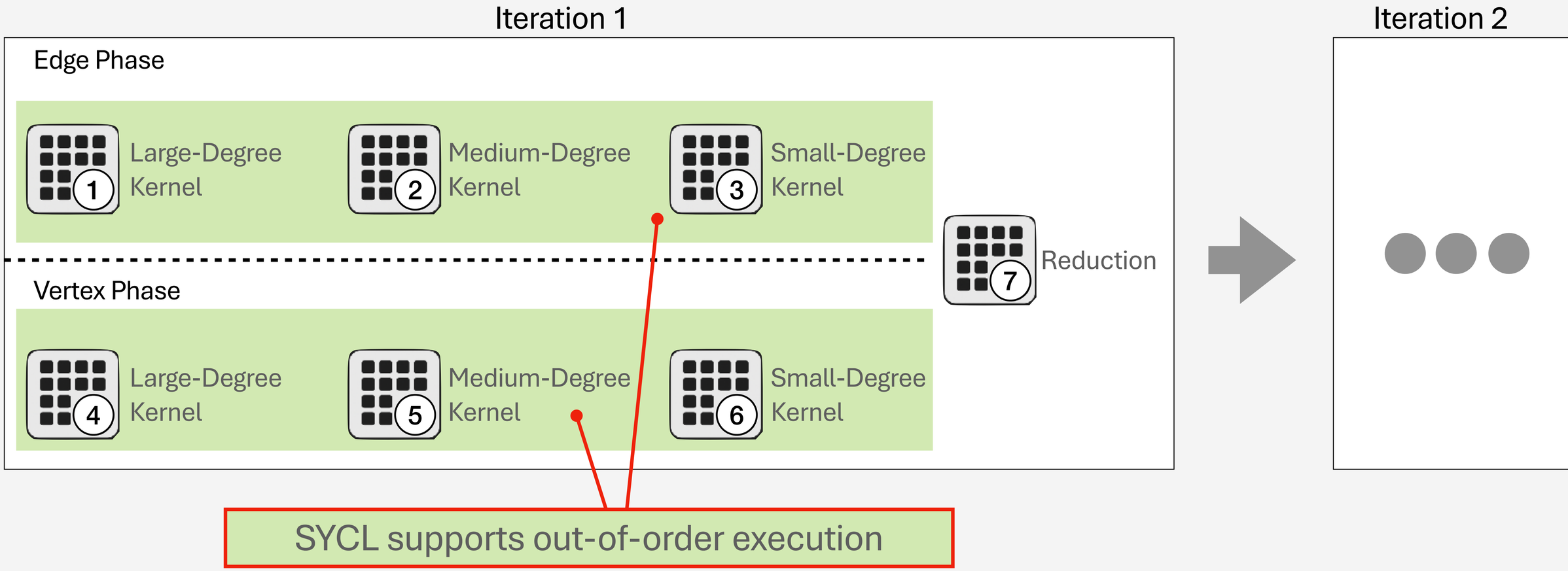
Summary Workflow



Summary Workflow

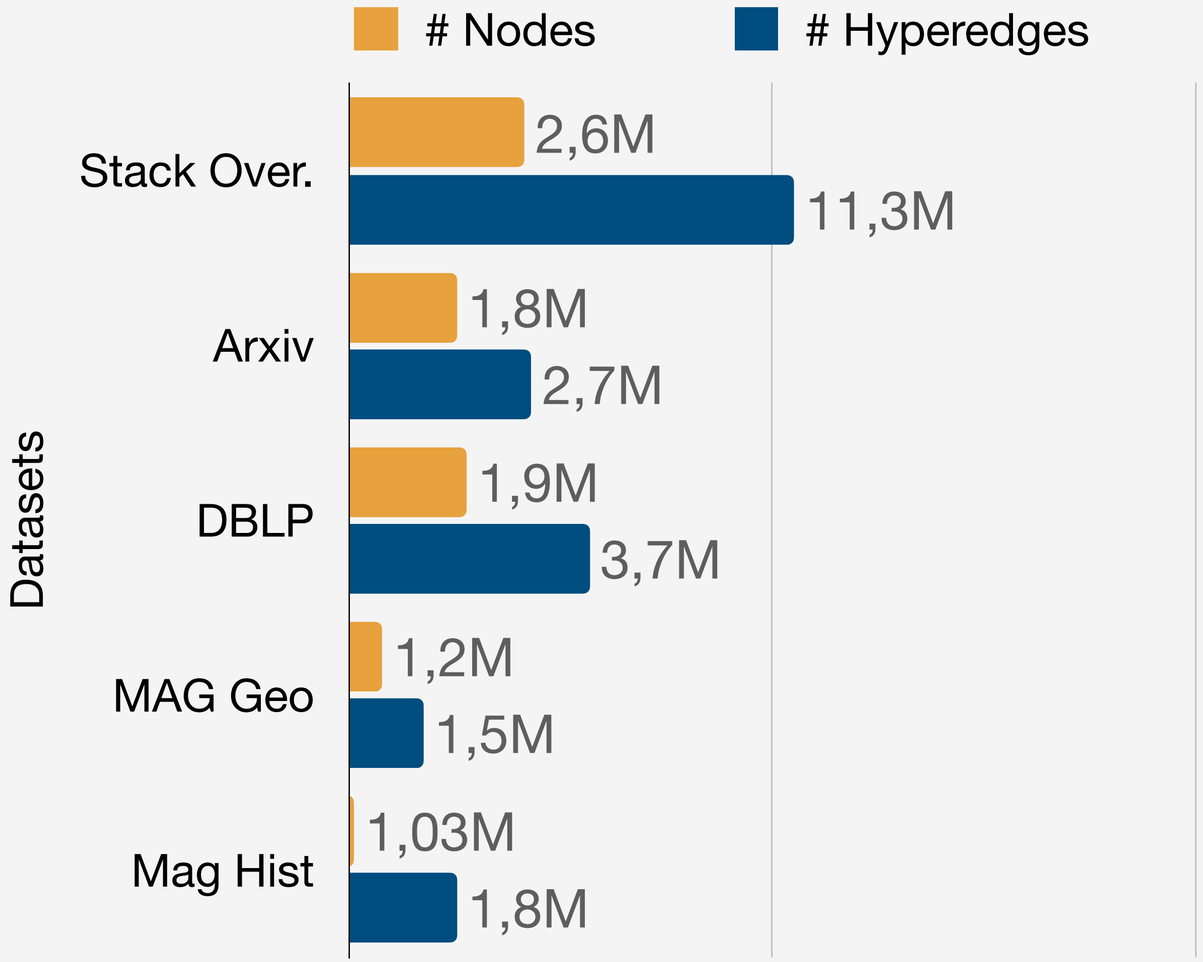


Summary Workflow

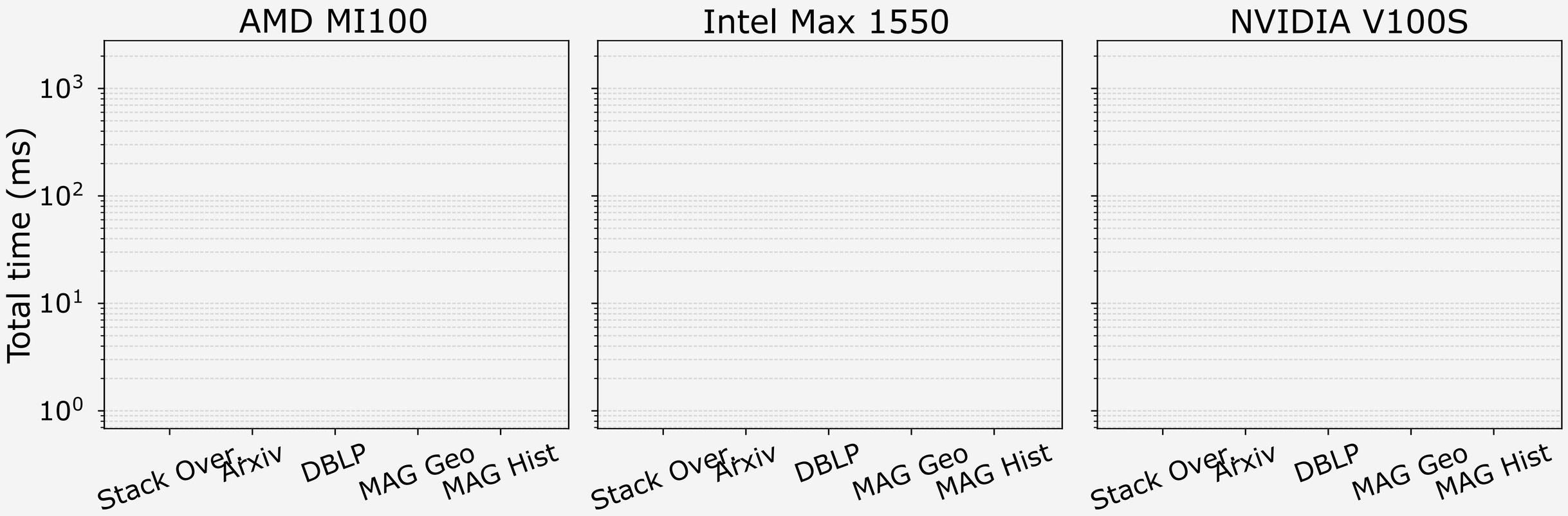


Experimental Setup

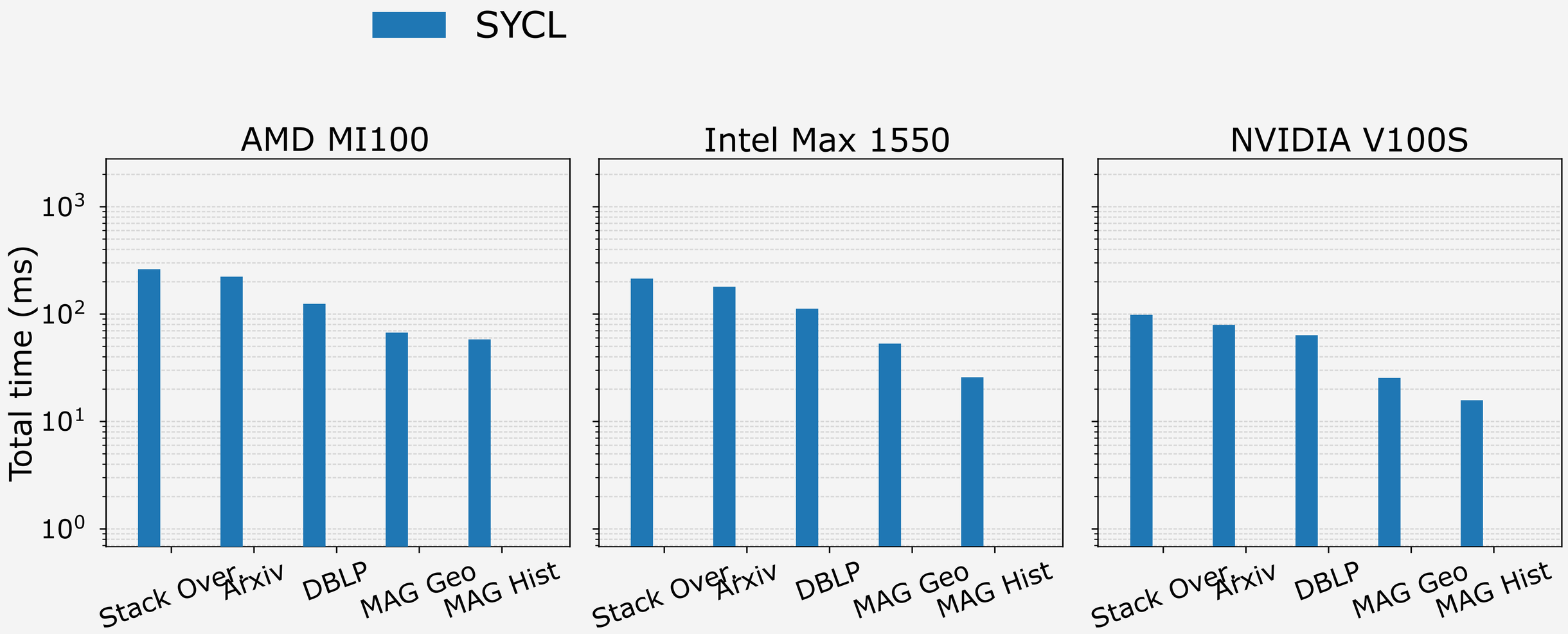
- Hardware
  - **AMD MI100**
  - **Intel Max 1550**
  - **NVIDIA V100S**
- Method
  - 20 runs per configuration
  - Median runtime reported



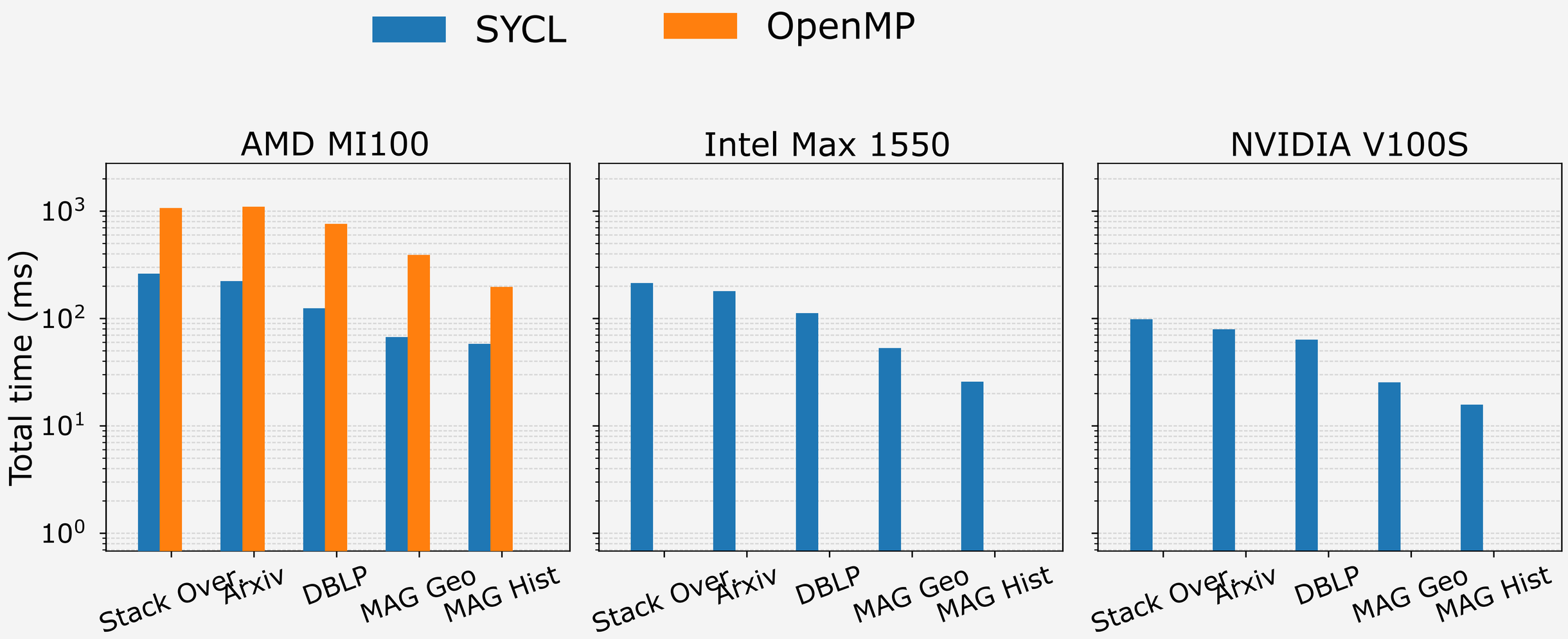
Runtime Analysis



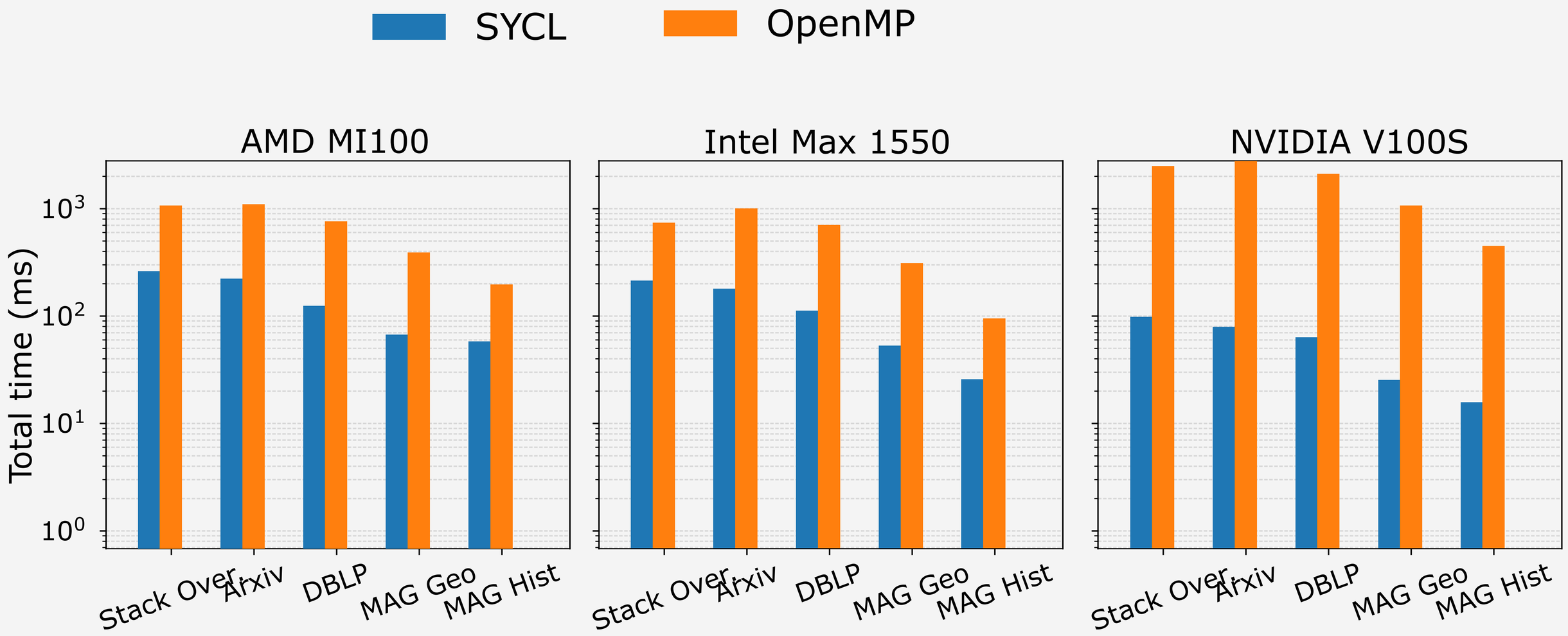
Runtime Analysis



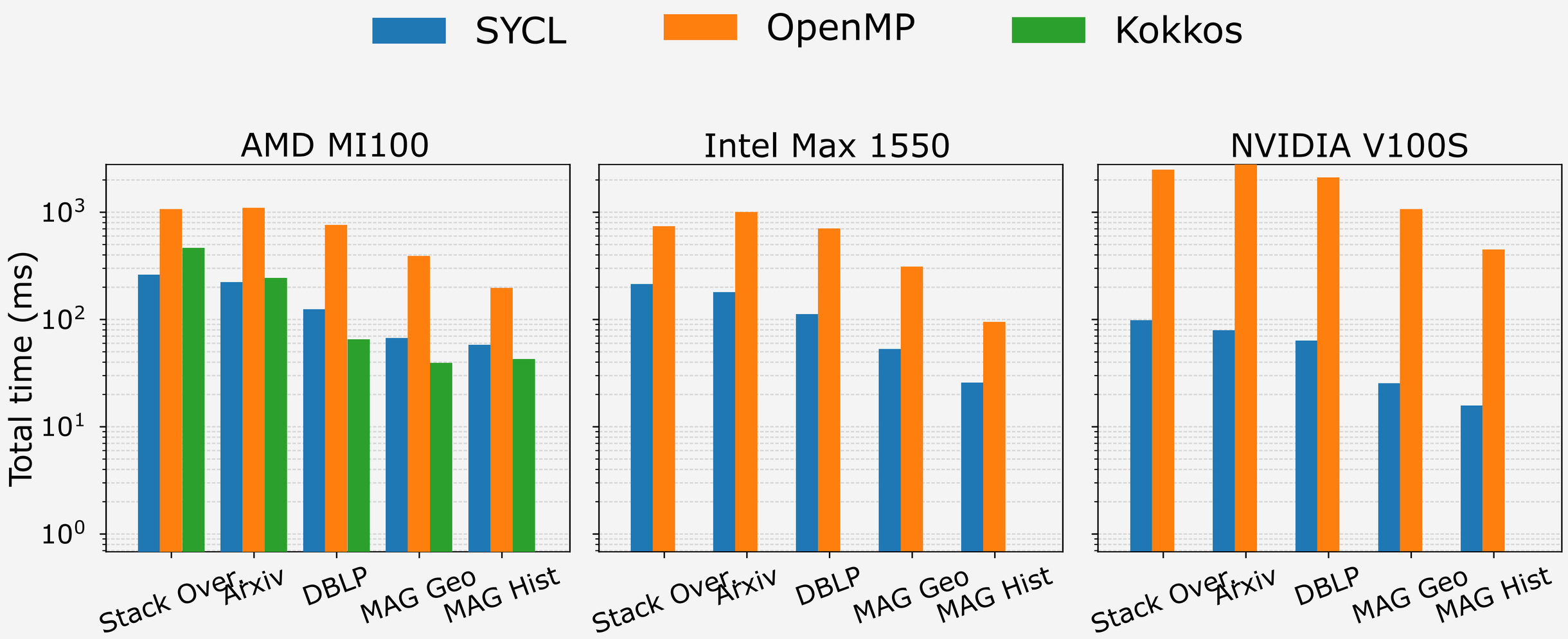
Runtime Analysis



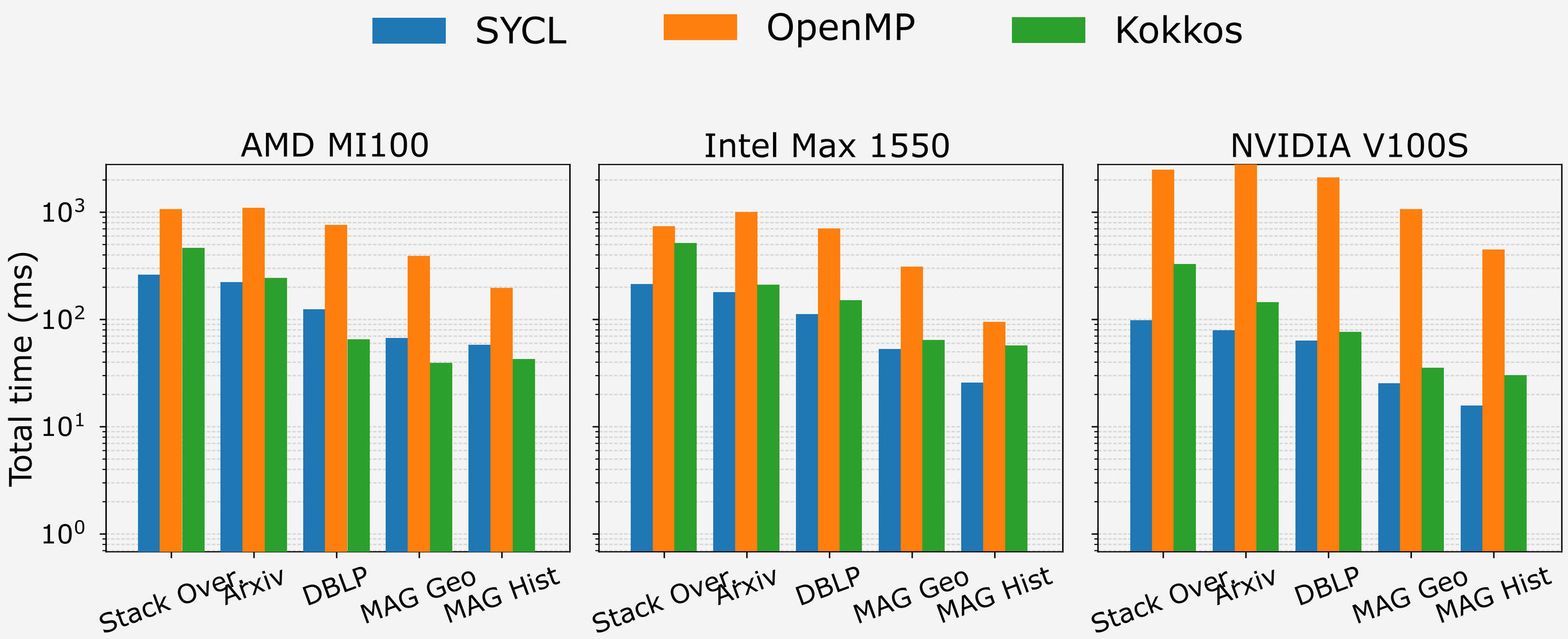
Runtime Analysis



Runtime Analysis



Runtime Analysis



Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

$$PP(A,P,H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

$$PP(A,P,H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

Application

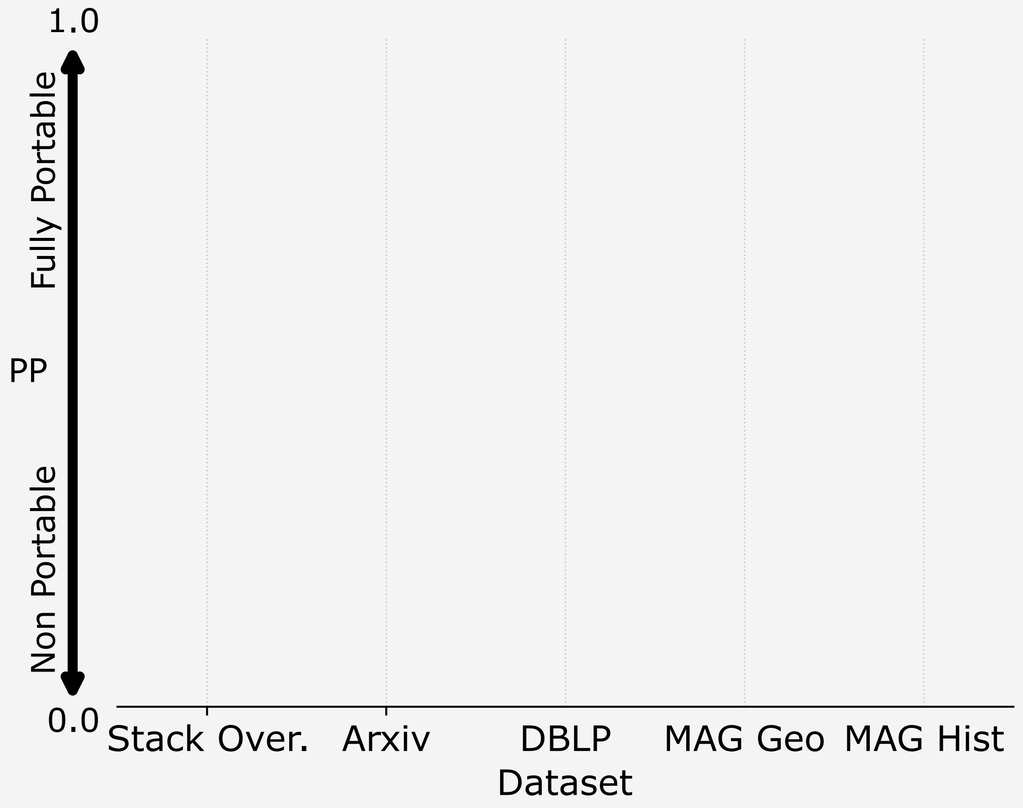
Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

Programming Model

$$PP(A, P, H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

Application



Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

Programming Model

$$PP(A, P, H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

Application

Devices

Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

Programming Model

$$PP(A, P, H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

Theoretical Roofline Peak

Application

Devices

Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

$$PP(A, P, H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

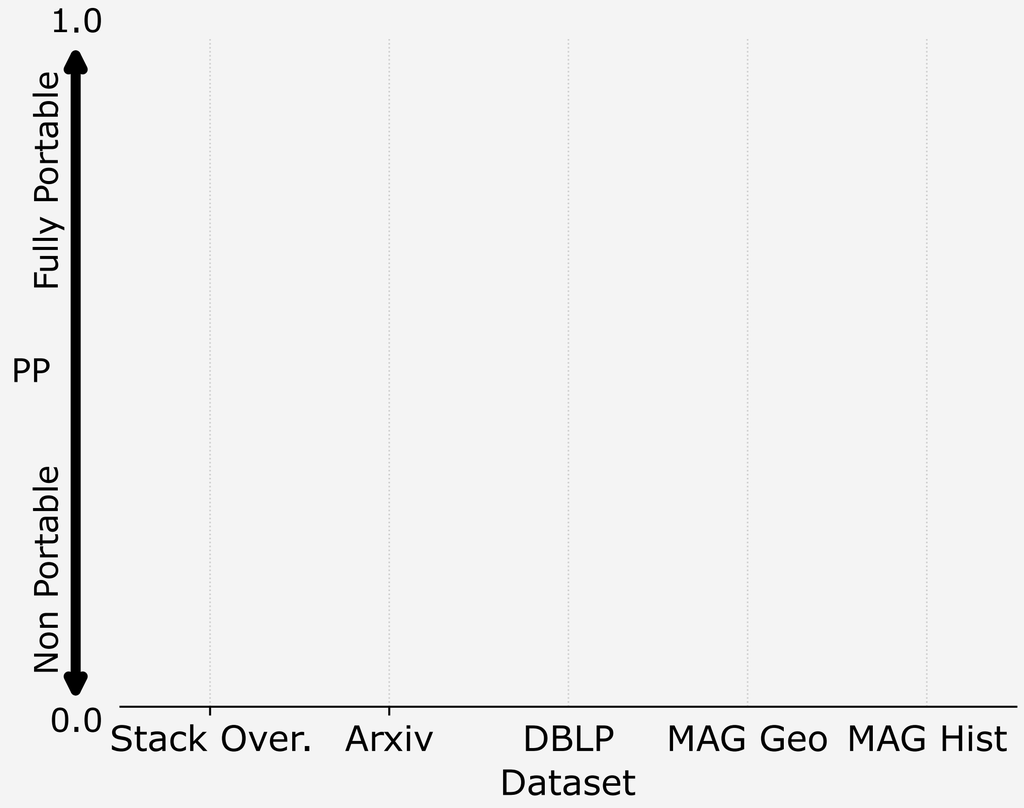
Programming Model

Application

Devices

Theoretical Roofline Peak

Achieved Roofline Efficiency



Performance Portability Analysis

- We used the Pennycook Metric to assess performance portability
- We consider the **instruction roofline model**

$$PP(A, P, H) = \frac{|H|}{\sum_{i \in H} \frac{\Lambda_i}{\lambda_i}}$$

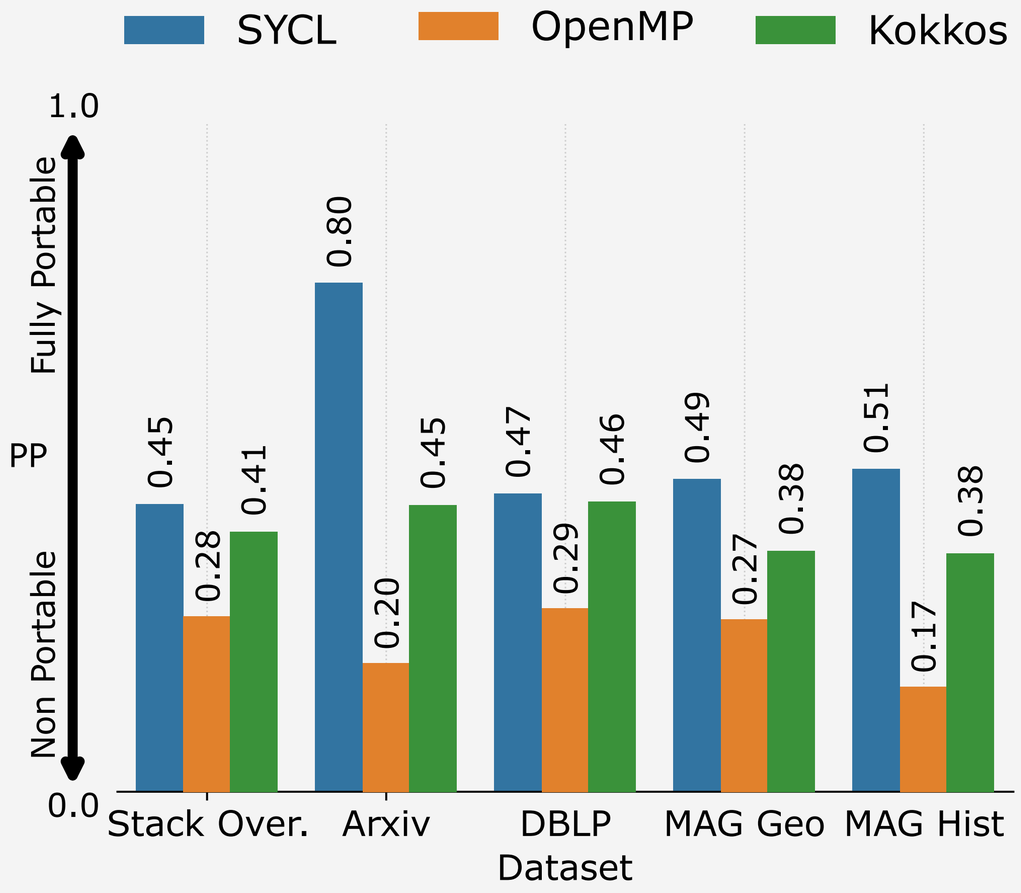
Programming Model

Application

Devices

Theoretical Roofline Peak

Achieved Roofline Efficiency



Evaluating Portable Programming Models for Hypergraph Label Propagation on GPUs

Antonio De Caro
University of Salerno
Fisciano, Italy

Dario De Maio
University of Salerno
Fisciano, Italy

Francesco Monzillo
University of Salerno
Fisciano, Italy

Alessia Antelmi
University of Turin
Turin, Italy

Biagio Cosenza
University of Salerno
Fisciano, Italy

***Abstract*—Understanding community formation is a fundamental task in network science, as it reveals the structural organization of complex networks and provides insights into the functional roles and interactions of their nodes. Recently, hypergraphs have emerged as a powerful framework for modeling high-order relationships in real-world systems, capturing multi-entity relations beyond traditional pairwise connections. However, efficiently processing hypergraphs on GPUs remains challenging due to their inherent sparsity and structural irregularity, leading to poor memory locality and load imbalance. Given that the world's most powerful supercomputers are equipped with GPUs from different vendors, such as AMD, Intel, and NVIDIA, a portable performance solution is essential to exploit these systems effectively without rewriting the entire codebase for each platform. In this work, we pursue this objective by evaluating three portable programming models, OpenMP, SYCL, and Kokkos, applied to the label propagation community detection algorithm for hypergraphs, examining their programmability and performance on heterogeneous GPU architectures.**

***Index Terms*—Label Propagation, Hypergraphs, Portable Programming Models, GPU**

I. Introduction

Community detection is the process of identifying groups of nodes in a network that are more densely connected to each other than to the rest of the network. Mining these local network structures can reveal the organizational principles and operational functions of a variety of real-world systems, ranging from news spreading to protein identification with similar biological functions to collaboration patterns [1]. Traditionally, complex systems have been successfully studied through graphs abstracting the underlying relations with nodes and edges connecting pairs of interacting components. However, many real-world dynamics are inherently high-order and cannot be described simply in terms of pairs [2]. Recently, hypergraphs have emerged as a powerful framework for naturally modeling such many-to-many relations [3].

A *hypergraph* is a generalization of a graph, where a (hyper)edge can connect an arbitrary number of nodes [4]. Such structures can easily abstract social systems where individuals interact in groups of any size (e.g., co-authorship collaboration network) as well as high-order relations in biological, ecological, and neuroscience systems [2]. Despite their powerful expressiveness, hypergraphs introduce significant computational and algorithmic challenges. Like graphs, handling hypergraphs on GPUs is particularly complicated due to their sparse and irregular structure [5]. Traditional representations such as incidence lists or incidence matrices are poorly suited to GPUs, as they require indirect and irregular memory accesses that are difficult to organize into contiguous, coalesced memory accesses. As a result, hypergraph workloads often exhibit uncoalesced memory access patterns and severe branch divergence, hindering GPU utilization and leading to uneven thread workload distribution. Designing efficient data layouts and parallel execution strategies is therefore non-trivial, especially when targeting diverse GPU architectures. Moreover, the increasing heterogeneity of systems in TOP500 [6]–featuring NVIDIA, AMD, and Intel accelerators–has further motivated the adoption of portable programming models such as SYCL, OpenMP target offload, and Kokkos to ensure performance between vendors while avoiding lock-in.

This work presents and evaluates three portable GPU implementations of the Label Propagation (LP) community detection algorithm designed for hypergraphs [7] using SYCL, OpenMP target offload, and Kokkos. Our analysis provides a comprehensive comparison of their programmability, performance, and performance portability across AMD, Intel, and NVIDIA GPUs, under a common optimization scenario. This study highlights the trade-offs among these models in handling irregular, memory-bound workloads and identifies key factors influencing their efficiency on heterogeneous architectures. The contributions of our work can be summarized as follows:

- We provide the first implementation of the Label Propagation algorithm for hypergraphs on GPUs using three portable programming models—SYCL, OpenMP target offload, and Kokkos;
- We introduce a uniform set of GPU optimizations, including coalesced hypergraph topology access, degree-aware phase decomposition, and shared-memory reuse, applied across all programming models;
- We offer a benchmark across AMD, Intel, and NVIDIA GPUs, collecting key hardware-level metrics and analyzing performance portability and productivity differences among SYCL, OpenMP, and Kokkos.

II. Background

A. Community Detection on Hypergraphs and the Label Propagation Algorithm

Let $H = (V, E)$ be a hypergraph where V is a finite set of *vertices*, and E is a set of non-empty subsets of V,

Evaluating Portable Programming Models for Hypergraph Label Propagation on GPUs

Antonio De Caro
University of Salerno
Fisciano, Italy

Dario De Maio
University of Salerno
Fisciano, Italy

Francesco Monzillo
University of Salerno
Fisciano, Italy

Alessia Antelmi
University of Turin
Turin, Italy

Biagio Cosenza
University of Salerno
Fisciano, Italy

***Abstract*—Understanding community formation is a fundamental task in network science, as it reveals the structural organization of complex networks and provides insights into the functional roles and interactions of their nodes. Recently, hypergraphs have emerged as a powerful framework for modeling high-order relationships in real-world systems, capturing multi-entity relations beyond traditional pairwise connections. However, efficiently processing hypergraphs on GPUs remains challenging due to their inherent sparsity and structural irregularity, leading to poor memory locality and load imbalance. Given that the world's most powerful supercomputers are equipped with GPUs from different vendors, such as AMD, Intel, and NVIDIA, a portable performance solution is essential to exploit these systems effectively without rewriting the entire codebase for each platform. In this work, we pursue this objective by evaluating three portable programming models, OpenMP, SYCL, and Kokkos, applied to the label propagation community detection algorithm for hypergraphs, examining their programmability and performance on heterogeneous GPU architectures.**

***Index Terms*—Label Propagation, Hypergraphs, Portable Programming Models, GPU**

I. Introduction

Community detection is the process of identifying groups of nodes in a network that are more densely connected to each other than to the rest of the network. Mining these local network structures can reveal the organizational principles and operational functions of a variety of real-world systems, ranging from news spreading to protein identification with similar biological functions to collaboration patterns [1]. Traditionally, complex systems have been successfully studied through graphs abstracting the underlying relations with nodes and edges connecting pairs of interacting components. However, many real-world dynamics are inherently high-order and cannot be described simply in terms of pairs [2]. Recently, hypergraphs have emerged as a powerful framework for naturally modeling such many-to-many relations [3].

A *hypergraph* is a generalization of a graph, where a (hyper)edge can connect an arbitrary number of nodes [4]. Such structures can easily abstract social systems where individuals interact in groups of any size (e.g., co-authorship collaboration network) as well as high-order relations in biological, ecological, and neuroscience systems [2]. Despite their powerful expressiveness, hypergraphs introduce significant computational and algorithmic challenges. Like graphs, handling hypergraphs on GPUs is particularly complicated due to their sparse and irregular structure [5]. Traditional representations such as incidence lists or incidence matrices are poorly suited to GPUs, as they require indirect and irregular memory accesses that are difficult to organize into contiguous, coalesced memory accesses. As a result, hypergraph workloads often exhibit uncoalesced memory access patterns and severe branch divergence, hindering GPU utilization and leading to uneven thread workload distribution. Designing efficient data layouts and parallel execution strategies is therefore non-trivial, especially when targeting diverse GPU architectures. Moreover, the increasing heterogeneity of systems in TOP500 [6]–featuring NVIDIA, AMD, and Intel accelerators–has further motivated the adoption of portable programming models such as SYCL, OpenMP target offload, and Kokkos to ensure performance between vendors while avoiding lock-in.

This work presents and evaluates three portable GPU implementations of the Label Propagation (LP) community detection algorithm designed for hypergraphs [7] using SYCL, OpenMP target offload, and Kokkos. Our analysis provides a comprehensive comparison of their programmability, performance, and performance portability across AMD, Intel, and NVIDIA GPUs, under a common optimization scenario. This study highlights the trade-offs among these models in handling irregular, memory-bound workloads and identifies key factors influencing their efficiency on heterogeneous architectures. The contributions of our work can be summarized as follows:

- We provide the first implementation of the Label Propagation algorithm for hypergraphs on GPUs using three portable programming models—SYCL, OpenMP target offload, and Kokkos;
- We introduce a uniform set of GPU optimizations, including coalesced hypergraph topology access, degree-aware phase decomposition, and shared-memory reuse, applied across all programming models;
- We offer a benchmark across AMD, Intel, and NVIDIA GPUs, collecting key hardware-level metrics and analyzing performance portability and productivity differences among SYCL, OpenMP, and Kokkos.

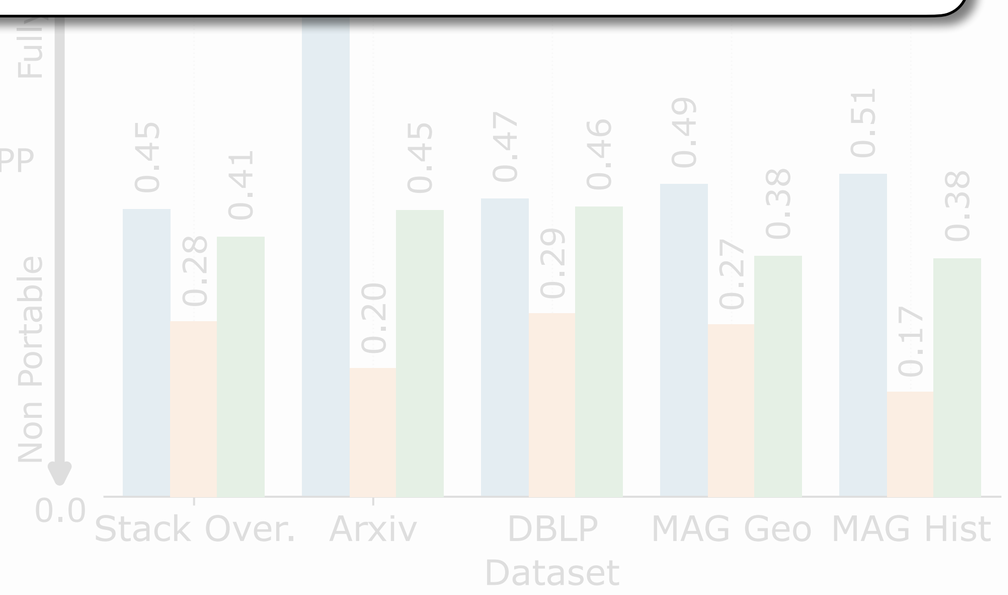
II. Background

A. Community Detection on Hypergraphs and the Label Propagation Algorithm

Let $H = (V, E)$ be a hypergraph where V is a finite set of *vertices*, and E is a set of non-empty subsets of V,

More Detailed Implementation

SYCL™ kokkos OpenMP



Evaluating Portable Programming Models for Hypergraph Label Propagation on GPUs

Antonio De Caro, *University of Salerno*, Fisciano, Italy
Dario De Maio, *University of Salerno*, Fisciano, Italy
Francesco Monzillo, *University of Salerno*, Fisciano, Italy
Alessia Antelmi, *University of Turin*, Turin, Italy
Biagio Cosenza, *University of Salerno*, Fisciano, Italy

***Abstract*—Understanding community formation is a fundamental task in network science, as it reveals the structural organization of complex networks and provides insights into the functional roles and interactions of their nodes. Recently, hypergraphs have emerged as a powerful framework for modeling high-order relationships in real-world systems, capturing multi-entity relations beyond traditional pairwise connections. However, efficiently processing hypergraphs on GPUs remains challenging due to their inherent sparsity and structural irregularity, leading to poor memory locality and load imbalance. Given that the world's most powerful supercomputers are equipped with GPUs from different vendors, such as AMD, Intel, and NVIDIA, a portable performance solution is essential to exploit these systems effectively without rewriting the entire codebase for each platform. In this work, we pursue this objective by evaluating three portable programming models, OpenMP, SYCL, and Kokkos, applied to the label propagation community detection algorithm for hypergraphs, examining their programmability and performance on heterogeneous GPU architectures.**

***Index Terms*—Label Propagation, Hypergraphs, Portable Programming Models, GPU**

I. Introduction

Community detection is the process of identifying groups of nodes in a network that are more densely connected to each other than to the rest of the network. Mining these local network structures can reveal the organizational principles and operational functions of a variety of real-world systems, ranging from news spreading to protein identification with similar biological functions to collaboration patterns [1]. Traditionally, complex systems have been successfully studied through graphs abstracting the underlying relations with nodes and edges connecting pairs of interacting components. However, many real-world dynamics are inherently high-order and cannot be described simply in terms of pairs [2]. Recently, hypergraphs have emerged as a powerful framework for naturally modeling such many-to-many relations [3].

A *hypergraph* is a generalization of a graph, where a (hyper)edge can connect an arbitrary number of nodes [4]. Such structures can easily abstract social systems where individuals interact in groups of any size (e.g., co-authorship collaboration network) as well as high-order relations in biological, ecological, and neuroscience systems [2]. Despite their powerful expressiveness, hypergraphs introduce significant computational and algorithmic challenges. Like graphs, handling hypergraphs on GPUs is particularly complicated due to their sparse and irregular structure [5]. Traditional representations such as incidence lists or incidence matrices are poorly suited to GPUs, as they require indirect and irregular memory accesses that are difficult to organize into contiguous, coalesced memory accesses. As a result, hypergraph workloads often exhibit uncoalesced memory access patterns and severe branch divergence, hindering GPU utilization and leading to uneven thread workload distribution. Designing efficient data layouts and parallel execution strategies is therefore non-trivial, especially when targeting diverse GPU architectures. Moreover, the increasing heterogeneity of systems in TOP500 [6]–featuring NVIDIA, AMD, and Intel accelerators–has further motivated the adoption of portable programming models such as SYCL, OpenMP target offload, and Kokkos to ensure performance between vendors while avoiding lock-in.

This work presents and evaluates three portable GPU implementations of the Label Propagation (LP) community detection algorithm designed for hypergraphs [7] using SYCL, OpenMP target offload, and Kokkos. Our analysis provides a comprehensive comparison of their programmability, performance, and performance portability across AMD, Intel, and NVIDIA GPUs, under a common optimization scenario. This study highlights the trade-offs among these models in handling irregular, memory-bound workloads and identifies key factors influencing their efficiency on heterogeneous architectures. The contributions of our work can be summarized as follows:

- We provide the first implementation of the Label Propagation algorithm for hypergraphs on GPUs using three portable programming models—SYCL, OpenMP target offload, and Kokkos;
- We introduce a uniform set of GPU optimizations, including coalesced hypergraph topology access, degree-aware phase decomposition, and shared-memory reuse, applied across all programming models;
- We offer a benchmark across AMD, Intel, and NVIDIA GPUs, collecting key hardware-level metrics and analyzing performance portability and productivity differences among SYCL, OpenMP, and Kokkos.

II. Background

A. Community Detection on Hypergraphs and the Label Propagation Algorithm

Let $H = (V, E)$ be a hypergraph where V is a finite set of *vertices*, and E is a set of non-empty subsets of V,

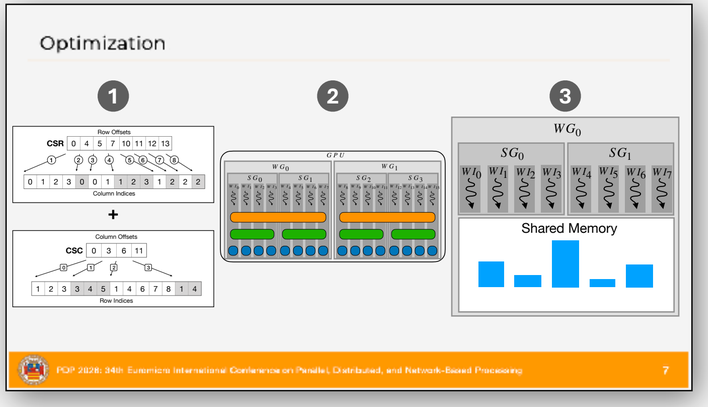
More Detailed Implementation

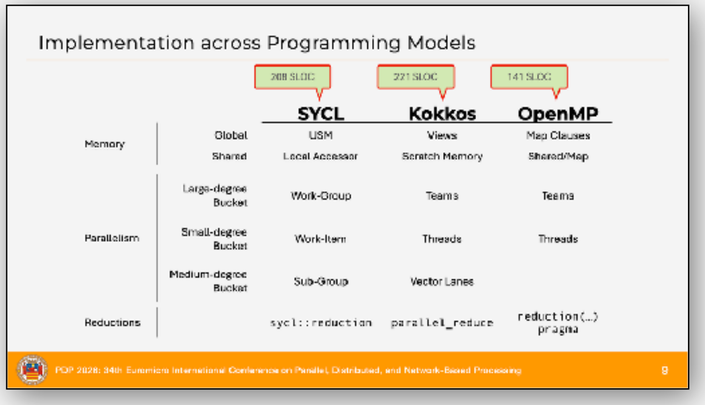
SYCL · kokkos · OpenMP

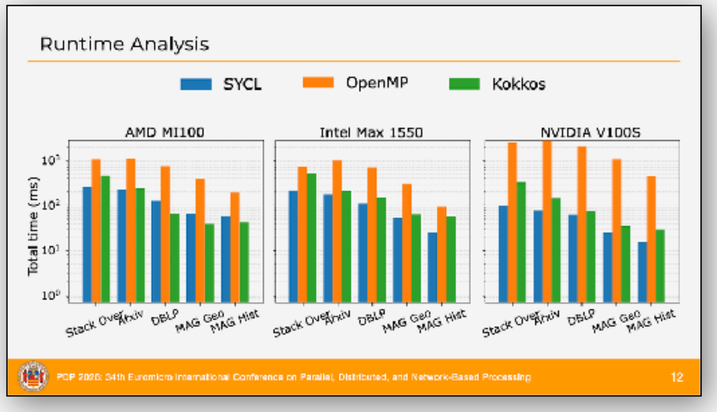
Hardware Metric Characterization

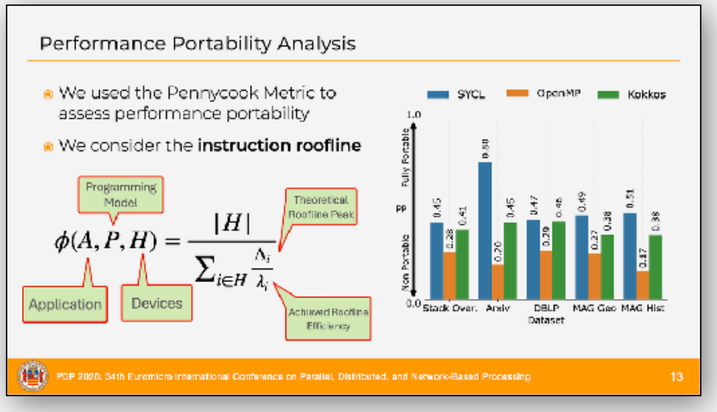
		Occupancy			Mem. Throughput			Comp. Throughput			Cache Util.		
		SYCL	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*
Stack Over.	*AMD MI100*	96.3	44.4	98.1	**87.0**	36.3	80.0	77.9	66.9	**83.4**	75.9	77.5	77.6
	Intel Max 1550	75.3	20.3	58.4	10.4	3.0	7.5	29.7	15.0	28.9	57.6	11.4	51.8
	NVIDIA V100S	**89.7**	24.6	63.9	45.1	27.7	41.0	22.2	16.1	20.0	20.5	32.5	24.9
Arxiv	*AMD MI100*	87.3	38.8	70.1	**96.4**	23.1	74.5	**94.9**	17.7	87.9	82.6	83.1	**83.2**
	Intel Max 1550	**92.6**	10.0	60.2	3.1	1.3	1.1	17.4	4.9	14.6	19.6	4.2	8.6
	NVIDIA V100S	90.4	24.9	61.9	79.9	20.3	21.2	51.9	8.4	9.6	35.8	41.1	33.2
DBLP	*AMD MI100*	84.4	37.3	**91.7**	**82.7**	33.6	73.6	85.7	61.9	**91.0**	**81.4**	79.4	79.9
	Intel Max 1550	63.1	15.4	54.1	3.8	1.3	1.5	18.9	12.2	19.5	25.6	8.6	14.4
	NVIDIA V100S	86.6	24.9	61.9	46.7	28.9	45.7	18.7	16.4	19.2	30.4	37.4	28.6
NAG: Geo	*AMD MI100*	82.6	41.2	**90.8**	**67.3**	32.8	58.5	85.7	49.1	**89.6**	81.8	82.1	**82.8**
	Intel Max 1550	46.9	15.9	44.9	0.0	1.1	0.0	15.4	11.6	19.6	28.9	7.0	13.8
	NVIDIA V100S	85.0	24.8	60.8	49.0	27.2	37.9	21.4	16.8	17.6	53.0	39.2	26.4
NAG: Hist.	*AMD MI100*	82.8	51.2	**85.0**	**50.7**	14.4	43.3	87.4	53.9	**92.2**	66.8	68.2	**69.0**
	Intel Max 1550	38.1	10.1	16.5	0.1	0.7	0.0	5.9	9.0	10.8	5.8	3.6	4.8
	NVIDIA V100S	83.2	24.4	50.3	30.6	16.6	17.6	30.4	15.6	14.0	61.2	40.8	29.3
Eventr	*AMD MI100*	79.8	**66.0**	**83.1**	**73.9**	16.3	39.4	85.7	8.5	**88.0**	82.9	**86.0**	**85.8**
	Intel Max 1550	23.8	4.7	18.6	0.0	0.3	0.0	4.4	3.3	6.2	0.8	0.8	0.7
	NVIDIA V100S	47.4	23.3	29.7	32.5	14.2	8.3	20.0	8.4	4.0	64.2	38.6	28.5

Conclusion









Reach us on Github

antdecaro@unisa.it