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,

called *hyperedges*. A *community* C in a hypergraph H is a subhypergraph H' whose vertices are densely connected within H' but sparsely connected to vertices in $H \setminus H'$. The task of identifying communities in hypergraphs generalizes the corresponding problem in graphs by incorporating the higher-order information encoded in hyperedges. Further, hyperedges themselves can also be assigned to communities. Formally, *community detection* aims to define a mapping function $\mathcal{F}$ that assigns each vertex $v \in V$ to at least one community $C_1, \ldots, C_k$, where k denotes the total number of communities discovered.

In this work, we focus on the Label Propagation community detection algorithm, which is widely recognized as efficient, scalable, and formally well defined [8]. Specifically, we consider the algorithm for hypergraphs proposed by Antelmi et al. [7], which generalizes the original algorithm introduced by Raghavan et al. [9] for graphs. The algorithm alternates between hyperedge and vertex update phases, allowing labels to propagate through higher-order connections in the hypergraph. During each iteration, information is exchanged between incident vertices and hyperedges until a stable labeling is reached. A GPU-oriented formulation of this two-phase propagation scheme, emphasizing its data movement and memory access characteristics, is described in Section III.

B. Portable Programming Models

The growing presence of heterogeneous GPU-based architectures in modern supercomputers–featuring devices from NVIDIA, AMD, and Intel–has made performance portability a primary concern for scientific computing. Developers aim to balance programmability, fine-grained control, and efficient execution without maintaining multiple vendor-specific codebases. To address this challenge, several portable programming models have emerged, offering abstractions that enable applications to target multiple GPUs with a single implementation. Despite syntactic differences, these models share common concepts: computations are expressed as kernels executed by lightweight threads organized into hierarchical groups (e.g., thread blocks or teams), and memory hierarchies include fast on-chip shared memory to accelerate data reuse.

SYCL [10] is a Khronos Group standard for single-source C++ programming on heterogeneous systems. It defines kernels launched on device queues, where *work-items* (threads) form *work-groups* (thread blocks) and hardware *subgroups* (warps or wavefronts) to exploit fine-grained SIMD-style parallelism. SYCL offers both a buffer–accessor model for dependency tracking and Unified Shared Memory for explicit pointer control, supporting local memory for efficient data reuse across backends such as Level Zero, CUDA, and HIP.

OpenMP target offload [11] extends its directive-based model to GPUs via constructs like `#pragma omp target teams distribute parallel for`. It enables incremental porting of CPU code by annotating loops and automates data transfers via *map* clauses. Parallelism is organized into hierarchical teams and threads, with optional shared memory and reductions introduced in recent versions, though explicit control over subgroup execution remains limited.

Kokkos [12] provides a hierarchical C++ abstraction that mirrors GPU execution through teams, vector lanes, and threads. Its `TeamPolicy` maps these levels to hardware-specific units (warps, wavefronts, subgroups) and supports team's scratch memory for on-chip data reuse. Kokkos targets multiple execution spaces (CUDA, HIP, SYCL, OpenMP), enabling portable yet fine-grained optimizations across architectures.

III. Algorithm Description

Given the heterogeneity of the programming models discussed, we introduce a unified abstraction that captures the fundamental GPU programming concepts underlying each model (see Algorithm 1), while remaining independent of their specific implementation.

At each iteration, the labels are alternately propagated between vertices and hyperedges until convergence or the maximum number of iterations is reached. The process begins with an *edge phase* (line 7), where all hyperedges are processed in parallel. Each thread or thread block handles a single hyperedge and computes the most frequent label among its incident vertices (lines 9–12). To effectively identify the majority label, each thread reserves a small histogram in shared memory (line 8) used to count the label occurrences. The resulting dominant label is then assigned to the corresponding hyperedge (line 14).

Once the hyperedge labels are updated, the algorithm proceeds with the *vertex phase* (line 16), which is also executed in parallel across all vertices. Each thread processes one vertex and examines the labels of the hyperedges to which it belongs (lines 18–20). As in the previous phase, a shared-memory histogram (line 17) is employed to identify the most frequent label among the neighboring hyperedges. If the new label differs from the vertex's previous assignment, the vertex label is updated (line 23), and a convergence flag is set to indicate that further iterations are required.

The alternation between these two phases continues until the relative proportion of changed vertex labels falls below the tolerance threshold t or the maximum iteration count is reached (lines 28–30). The parameter t thus serves as a convergence criterion, ensuring the algorithm halts when subsequent updates become negligible, thereby avoiding unnecessary iterations.

In this formulation, outer-level parallelism is achieved across vertices and hyperedges, while the loops over their local incident elements are executed sequentially within each thread. The use of shared memory minimizes global memory traffic and accelerates the computation of label frequencies, while atomic operations ensure consistent updates to the *changes* variable (line 24) during concurrent access. The final output consists of stable vertex and hyperedge label assignments representing the propagated community structure within the hypergraph. Details concerning each programming model implementation are highlighted in the next sections.

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

Algorithm 1: Hypergraph Label Propagation (HLP) — GPU abstraction

Result: List of hyperedge and vertex labels
Input: $H = (V, E)$, V_L, $\mathcal{L}$, *max_iter*, t tolerance threshold

1 vlabels $\leftarrow$ map$\{Int, Int\}$;
2 helabels $\leftarrow$ map$\{Int, Int\}$;
3 changes $\leftarrow 0$; iter $\leftarrow 0$;
4 stop $\leftarrow false$;
5 **while** $\neg$*stop* ***and*** *iter* $<$ *max_iter* **do**
6   stop $\leftarrow true$;
7   **for** $e \in E$ **do in parallel**
8   **shared** hist$[|\mathcal{L}|] \leftarrow 0$;
9   **for** $u \in e$ **do**
10   **if** *vlabels*$[u]$ *defined* **then**
11   **increment** hist[vlabels$[u]$]
12   **end**
13   **end**
14   helabels$[e] \leftarrow \arg\max_c$ hist$[c]$;
15   **end**
16   **for** $v \in V$ **do in parallel**
17   **shared** hist$[|\mathcal{L}|] \leftarrow 0$;
18   **for** $e \ni v$ **do**
19   **increment** hist[helabels$[e]$]
20   **end**
21   label $\leftarrow \arg\max_c$ hist$[c]$;
22   **if** *label* $\neq$ *vlabels*$[v]$ **then**
23   vlabels$[v] \leftarrow$ label;
24   **atomicAdd**(changes, 1);
25   stop $\leftarrow false$;
26   **end**
27   **end**
28   **if** *changes*$/|V| < t$ **then**
29   stop $\leftarrow true$;
30   **end**
31   changes $\leftarrow 0$;
32   iter $\leftarrow$ iter $+ 1$;
33 **end**
34 **return** vlabels

IV. Optimizations Overview

A. Hypergraph Representation

To optimize memory usage, we adopt a hybrid CSR+CSC layout [13] as shown in Figure 1. The Compressed Sparse Row (CSR) format stores, for each node v, the list of incident hyperedges. In contrast, the Compressed Sparse Column (CSC) stores, for each hyperedge, the set of nodes it contains. This dual representation supports both vertex-to-hyperedge and hyperedge-to-vertex traversals and ensures coalesced memory access to contiguous elements within each incidence list, improving memory efficiency and GPU throughput.

Such bidirectional access is essential for the two-phase structure of the Algorithm 1. During the *edge phase*, each hyperedge iterates over its incident vertices to aggregate their current labels, which requires fast access from hyperedges to nodes. In contrast, in the *vertex phase*, each vertex updates its label based on the labels of its incident hyperedges, demanding traversal in the opposite direction. By explicitly storing both mappings in CSR (node-to-hyperedge) and CSC (hyperedge-to-node) formats, we avoid expensive on-the-fly transpositions and enable efficient GPU-parallel executions across both phases.

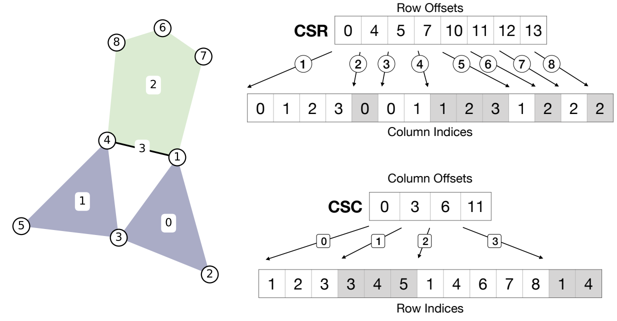

Fig. 1. CSR+CSC Layout.

B. Phase Decomposition and Hierarchical Parallelism

Both the *edge phase* and the *vertex phase* of Algorithm 1 operate on hyperedges and vertices with highly irregular incidence sizes. Some hyperedges may contain hundreds or thousands of nodes, while others consist of only a few; similarly, certain vertices participate in many hyperedges, whereas others appear in very few. Executing all elements using a single, uniform parallel strategy leads to severe load imbalance and high thread divergence on GPU architectures.

To overcome this, we first perform a preprocessing step in which we compute degree-based groups for both hyperedges and vertices. Elements are bucketed according to the number of incident nodes (for hyperedges) or incident hyperedges (for vertices), as shown in Figure 2. We then decompose both the *edge phase* and the *vertex phase* into three execution steps, where each step processes a different group. Each step is implemented as a separate GPU kernel, allowing the execution configuration to be tailored to the size of each group.

Groups are assigned to different levels of the GPU's hierarchical execution model (in SYCL terminology): large-degree elements are processed by an entire work-group, medium-degree elements are handled at the sub-group level, and small-degree elements are executed by individual work-items. When processing at the work-group or sub-group level, work-items operate cooperatively over contiguous segments of the CSR and CSC, enabling coalesced global memory accesses.

By aligning computational effort with degree-based grouping and mapping it to appropriate levels of parallelism, we achieve better load balancing, increase hardware occupancy, reduce synchronization overhead, and minimize thread divergence during both phases of the algorithm. However, not all programming models expose the same degree of hierarchical parallelism or provide equivalent control over work groups, subgroups, and individual work items.

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

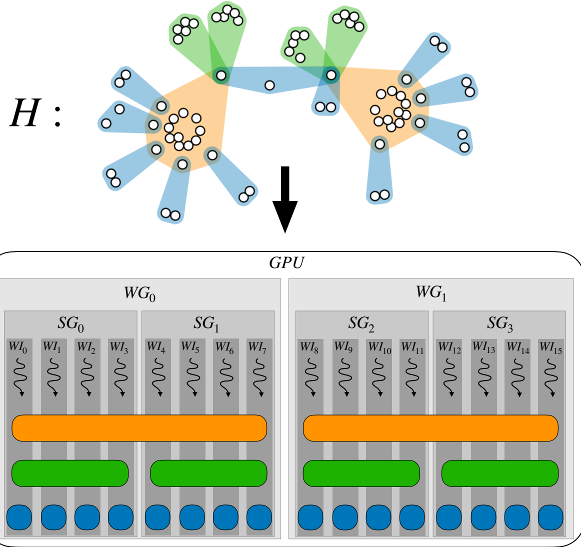

Fig. 2. Phase Decomposition example.

C. Local (Shared) Memory Usage

To further improve performance, we exploit local (shared) memory to reduce the number of global memory accesses during the label-aggregation steps in both phases. Since each group in our three-step execution strategy exhibits different levels of parallel cooperation, we adapt the use of local memory accordingly. In the first step, where large-degree elements are processed by an entire work-group, threads collaboratively update a single shared-memory buffer that stores the intermediate label histogram, which is then reduced to determine the dominant label. In the second step, medium-degree elements are assigned to subgroups; in this case, the local memory allocation is larger, as a distinct histogram buffer must be reserved for each subgroup within the work group to allow independent reductions to proceed without interference. Finally, in the third step, where small-degree elements are processed by individual work-items, each thread is given an exclusive portion of local memory to maintain its own label counts and directly retrieve the maximum label without requiring any synchronization.

V. Implementation on Different Programming Models

This section describes the approach adopted to implement Algorithm 1 across the programming models and how each of those implements the optimizations described in Section IV.

A. SYCL

The SYCL implementation uses *Unified Shared Memory* (USM) to explicitly allocate device memory. The dual CSR and CSC arrays describing the hypergraph are flattened on the host and copied into device-accessible USM allocations.

Each iteration of the algorithm launches the two phases (edge and vertex) and further decomposes them into three degree-based buckets. Each bucket is executed through an independent kernel submission to a *SYCL out-of-order queue*. This configuration is crucial for performance, as it allows kernels in the same phase to execute concurrently, improving GPU utilization and overlapping computation across buckets, since each edge is independent of the others.

Large-degree elements are assigned to entire work-groups, medium-degree to sub-groups, and small-degree to individual work-items using `nd_range` configurations. Within each kernel, threads cooperatively accumulate label occurrences in shared memory through a `sycl::local_accessor`. For the sub-group case, the accessor is partitioned to avoid conflicts between sub-groups, and partial results are combined using subgroup collectives. Updates to the global label arrays rely on `sycl::atomic_ref` to ensure correctness under concurrent writes.

Convergence detection is performed at the end of the vertex phase using a `sycl::reduction` over all threads to accumulate the number of label changes across vertices. The resulting scalar value is transferred back to the host to determine whether an additional iteration is required.

B. OpenMP Target Offload

The OpenMP implementation adopts a directive-based approach, using `#pragma omp target` constructs to offload computation to the GPU. The CSR and CSC arrays are transferred through `map(tofrom:)` clauses, avoiding explicit memory-management calls and keeping the host and device data consistent across iterations. Each propagation phase (edge and vertex) is executed as a `target teams distribute parallel for` construct, where teams correspond to GPU thread blocks and threads to individual device lanes.

Unlike SYCL and Kokkos, OpenMP currently does not expose explicit sub-group or SIMD-level parallelism on GPUs. Consequently, medium-degree elements, which could otherwise benefit from warp- or wavefront-level cooperation, are processed at the thread level. For this reason, each phase is decomposed into two kernel launches, corresponding to the large- and small-degree buckets, respectively.

Within each kernel, a buffer in local memory is allocated using `#omp parallel shared` to store temporary label histograms. However, dynamic local memory allocation is not consistently supported across all backends, as it is available only on Intel GPUs. Thus, to ensure portability across backends, a fixed-size static buffer is used. Threads within a team cooperatively update this buffer, relying on `#pragma omp barrier` for synchronization. For the small bucket, the scratch memory is subdivided per thread to avoid contention, and atomic updates to the global counter of label changes are handled using `#pragma omp atomic`.

C. Kokkos

The Kokkos backend expresses label propagation with hierarchical team parallelism and device-resident views, which flattens the CSR and CSC data into `Kokkos::Views` allocated in the selected execution space so the same source compiles for NVIDIA, AMD, or Intel targets.

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

TABLE I
SET OF REAL-WORLD DATASETS USED FOR EXPERIMENTS. $|V|$ DENOTES THE NUMBER OF NODES, $|E|$ THE NUMBER OF HYPEREDGES, $|E^*|$ THE NUMBER OF UNIQUE HYPEREDGES, AND s_{max} THE MAXIMUM HYPEREDGE SIZE.

Name	$\|V\|$	$\|E\|$	$\|E^*\|$	s_{max}
Stack Overflow [14]	2.675.969	11.305.356	9.705.575	67
Arxiv	1.821.977	2.765.236	1.986.653	2811
DBLP [14]	1.930.378	3.700.681	2.467.389	280
MAG Geo [14], [15]	1.261.129	1.591.166	1.204.704	284
MAG Hist [14], [15]	1.034.876	1.813.147	896.062	925
Eventer [16]	77.343	350.460	131.621	620
NDC [14]	5.556	112.919	10.273	187
Senate [17]	294	29.157	21.721	99

Each propagation phase is decomposed into three kernel launches. Computation within each launch follows the Kokkos hierarchical execution model, where teams, vector lanes, and threads handle the large-, medium-, and small-degree buckets, respectively. The edge phase is implemented using `Kokkos::parallel_for` with a `TeamPolicy`, meanwhile the vertex phase adopts the same structure but uses a `Kokkos::parallel_reduce` to accumulate the total number of vertex-label changes for convergence detection. This configuration allows the `league_size`, `team_size`, and `vector_length` parameters to be tuned per bucket, ensuring balanced occupancy and efficient resource utilization across architectures.

Within each kernel, temporary label histograms are allocated in local memory through `team.team_scratch(0)`. When processing large- or medium-degree buckets, concurrent updates to the same histogram entries are protected using `Kokkos::atomic_fetch_add` operations, while for smaller buckets, each thread maintains a private histogram in scratch memory, eliminating the need for atomic synchronization. The convergence condition is evaluated on the host after each vertex phase based on the total number of vertex label changes.

VI. EXPERIMENTAL EVALUATION

A. Datasets and Hardware Setup

Table I lists the datasets used in our experimental suite. These datasets have been collected from the XGI-DATA hypergraph repository [18], and they cover a broad range of domains and structural properties, providing a representative benchmark for evaluating heterogeneous GPU performance. The largest datasets (*Stack Overflow*, *Arxiv*, and *DBLP*) model co-occurrence and collaboration patterns in online and academic networks, featuring millions of vertices and hyperedges of different sizes. The *MAG-Geo* and *MAG-Hist* datasets originate from the Microsoft Academic Graph and exhibit highly skewed degree distributions, which stress memory bandwidth and expose load imbalance in the propagation phases. Smaller datasets such as *Eventer*, *NDC*, and *Senate* capture biomedical and social interactions and exhibit greater structural irregularity and dense local connectivity despite their limited size. We incorporated these smaller datasets to also evaluate the performance of each programming model on relatively small workloads. Tests were performed on three heterogeneous GPU systems, summarized in Table II. The SYCL implementation was compiled using oneAPI v2025.2. For OpenMP, we employed LLVM 17.0.6 on NVIDIA and AMD systems, and the `icpx` compiler from oneAPI v2025.2 for the Intel platform. The Kokkos implementation was built with Kokkos v4.7, using the CUDA, HIP, and SYCL execution backends corresponding to each device.

TABLE II
HARDWARE SETUP OF THE DIFFERENT ARCHITECTURES EMPLOYED IN THE EXPERIMENTS.

Vendor	GPU	VRAM	Backend
AMD	MI100	32GB	ROCm v7.0 and HIP v6.1
Intel	Max 1550	128GB	LevelZero v2025.2
NVIDIA	Tesla V100S	32GB	CUDA v12.3

B. Runtime Analysis

To evaluate our implementations, we randomly generated node labels for each dataset using a fixed seed as input to the random number generator. Figure 3 reports the median runtime over twenty executions for each hardware platform and programming model across all datasets in Table I. Table III presents the corresponding hardware metrics for each GPU on each dataset. We collected these metrics using `rocprof` for the AMD MI100, `VTune` for the Intel Max 1550, and `ncu` for the NVIDIA V100.

On the AMD MI100 GPU, SYCL achieves the best performance on the largest datasets, such as *Stack Overflow* and *Arxiv*, while Kokkos performs slightly better on the smaller ones. This behavior can be attributed to SYCL's use of an out-of-order queue, which allows concurrent execution of multiple kernels within a propagation phase, improving device utilization on large workloads. In particular, the *Stack Overflow* and *Arxiv* datasets yield larger sizes for the *large* and *medium* buckets per phase, increasing parallelism and favoring SYCL's out-of-order execution over Kokkos. On AMD, SYCL also achieves the highest memory throughput–up to 87% of theoretical bandwidth.

On the Intel Max 1550 GPU, SYCL consistently achieves the best performance across all datasets. This advantage is mainly due to Kokkos's reliance on SYCL as its execution backend on Intel GPUs, which introduces an additional software layer that increases runtime overhead. As a result, Kokkos exhibits lower occupancy than native SYCL despite similar computational patterns. Interestingly, OpenMP achieves relatively better performance on this platform than on NVIDIA or AMD GPUs, benefiting from the tighter integration of the OpenMP offload runtime within the oneAPI compiler stack. Nevertheless, its overall efficiency remains limited by reduced cache utilization and modest instruction throughput compared to SYCL. In addition, the lower measured memory throughput on Intel GPUs is mainly attributable

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

TABLE III
PEAK VALUES EXPRESSED IN PERCENTAGES OF HARDWARE PERFORMANCE METRICS FOR ALL HARDWARE PLATFORMS FOR EACH PROGRAMMING MODEL, EVALUATED ON THE MOST REPRESENTATIVE DATASETS.

		Occupancy			**Mem. Throughput**			**Comp. Throughput**			**Cache Util.**		
		SYCL	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*	*SYCL*	*OpenMP*	*Kokkos*
Stack Over.	*AMD MI100*	86.7	44.4	78.1	**87.0**	36.3	80.0	77.9	66.9	**83.4**	75.9	77.5	**77.6**
	Intel Max 1550	75.5	20.3	58.4	10.4	3.0	7.5	29.7	15.0	28.9	57.6	11.4	31.8
	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	87.0	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**	32.6	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	29.9	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	19.1	10.2	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	50.6	16.6	17.6	30.4	15.6	14.0	61.2	46.8	29.3
Eventer	*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	6.5	20.0	8.4	4.0	64.2	73.6	76.5

to their significantly higher peak memory bandwidth (3.2 TB/s, compared to 1.2 TB/s for the MI100 and 1.13 TB/s for the V100S), which reduces the relative utilization ratio.

On the NVIDIA V100S, SYCL consistently outperforms both OpenMP and Kokkos across all datasets, with Kokkos systematically achieving higher efficiency than OpenMP. This trend aligns with the hardware metrics, as SYCL reaches up to 90% occupancy and over 45% of peak memory throughput, while Kokkos maintains balanced compute and cache utilization. OpenMP, instead, shows the lowest occupancy and compute throughput due to the absence of subgroup-level parallelism and less efficient thread scheduling on the CUDA backend. On the smallest dataset (*Senate*), however, SYCL exhibits slightly higher overhead than Kokkos, likely due to the fixed launch cost of its out-of-order queue and synchronization mechanisms, which become more pronounced when the total workload is insufficient to saturate the GPU.

In conclusion, SYCL consistently achieves the highest occupancy across all programming models on the largest datasets, while OpenMP exhibits the lowest. This difference is primarily due to OpenMP's lack of SIMT- and vector-level parallelism, which limits its ability to efficiently process *medium-sized* buckets.

C. Performance Portability Evaluation

To quantitatively assess performance portability, we adopt the metric proposed by Pennycook *et al.* [19]. The metric, denoted as PP, is the harmonic mean of the roofline efficiencies achieved across the set of target platforms H:

$$PP(A,P,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{\Lambda_i}{\lambda_i}}, & \text{if } \lambda_i > 0,\ \forall i \in H \\ 0, & \text{otherwise} \end{cases}, \quad (1)$$

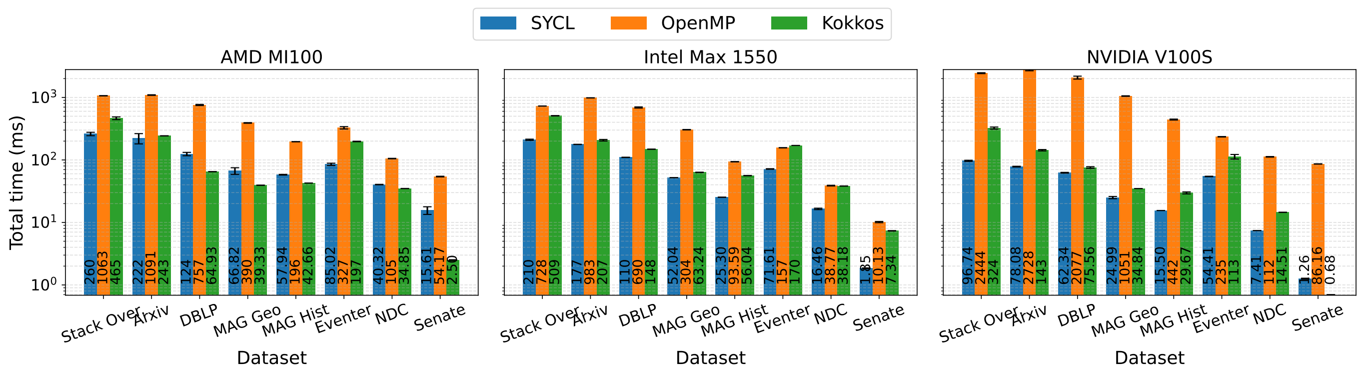

Fig. 3. Median of the runtimes of the HLP algorithm on AMD MI100, Intel Max 1100, and NVIDIA V100S for each programming model implementation.

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

where A is the application, P the programming model, and H the set of devices. For each platform $i \in H$, λ_i is the achieved roofline efficiency and Λ_i the corresponding peak. In this work, both values are derived from the *instruction roofline model* [20], which measures efficiency in terms of sustained instruction throughput rather than floating-point operations. This model is better suited to the HLP kernel, which is dominated by memory operations. The resulting efficiency λ_i/Λ_i therefore captures how closely each implementation approaches the instruction-level roofline bound.

Figure 4 reports the performance portability (PP) values obtained across the most significant datasets and calculated according to Equation 1. A value of $PP = 0$ indicates that the application fails to execute or achieves poor hardware usage on at least one platform, whereas $PP = 1$ represents perfect performance portability, with the application achieving peak performance on all platforms. Results indicate that SYCL consistently achieves the highest PP across all datasets, with a value ranging from 45% to 80%, followed by Kokkos with 38% to 46% and OpenMP from 17% to 29%. Both the figure and the reported values exclude the smallest datasets, which exhibit limited GPU utilization.

D. Productivity Analysis

We compared the programmability of SYCL, OpenMP, and Kokkos using code size measured in terms of *Source Lines of Code* (SLOC), defined as the number of non-empty, non-comment lines of source [21]. OpenMP target offload provides the most compact implementation (141 SLOC) thanks to pragma-based directives and implicit memory management. SYCL requires slightly more code (208 SLOC) due to explicit USM allocation and queue orchestration, but offers finer control over kernel scheduling and memory hierarchy. Kokkos is the most verbose (221 SLOC), mainly because of team and scratch management. Overall, OpenMP minimizes development effort, while SYCL and Kokkos expose richer tuning knobs and a better portability–control balance.

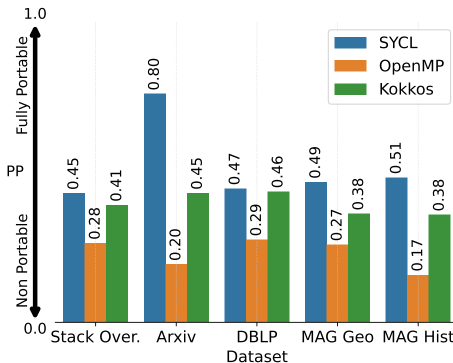

Fig. 4. Performance portability (PP) of SYCL, OpenMP, and Kokkos across the evaluated datasets.

VII. Related Work

Compared to graphs, hypergraph algorithms must explicitly handle high-order interactions encoded by hyperedges, which significantly affect algorithm design and performance on parallel architectures. To address these challenges, several frameworks have been developed to extend traditional graph analytics to hypergraphs. Notable examples include HyperX [22], a distributed framework built in Apache Spark, and Hygra [23], a shared-memory system that extends Ligra [24] for parallel hypergraph algorithms. Despite this increasing interest in hypergraph learning and GPU acceleration [3], research on classical hypergraph algorithms, such as LP, remains largely confined to CPU-based implementations [7], [25]–[27].

Specifically focusing on the task of community detection, the ever-increasing availability of data has led to massive, noisy, and dynamic networks; this growth, in turn, has driven the field toward approaches that favor scalability, adaptability, robustness, and simplicity (e.g., having as few tunable parameters) [8]. In this context, the LP algorithm [9] has gained widespread popularity due to its efficiency, conceptual clarity, and ease of implementation. Despite its simplicity, LP has more than 13k variants designed for graphs [8].

Several studies have proposed parallel and GPU-accelerated LP implementations to further enhance its performance on large-scale networks. For instance, *ν-LP* [28] adapts LP for SIMT architectures with an asynchronous scheme and a "Pick-Less" heuristic to improve convergence stability, combined with efficient per-vertex hash tables for label counting. Earlier GPU implementations, such as those by Mišić et al. [29] and Ye et al. [30], emphasize memory coalescing, and reduced host–device transfers to achieve high throughput. Fiscarelli and Brust [31] further improved stability using memory-augmented LP that retains historical label states. Although frameworks such as CUDA, SYCL, Kokkos, and OpenMP-offload enable GPU execution, most existing evaluations focus on structured, bandwidth-bound, or stencil-based kernels [32], while performance portability for irregular, topology-driven algorithms, such as hypergraph label propagation, remains underexplored. Recent works, such as HyperG [33], address irregularity by introducing balanced coarsening and refinement strategies to manage varying vertex degrees and uneven workloads during partitioning.

To the best of our knowledge, our work provides the first GPU implementation of the LP algorithm for hypergraphs and the first comparative study assessing its productivity, performance, and performance portability across SYCL, OpenMP, and Kokkos on GPUs from different hardware vendors.

VIII. Conclusion

In this work, we presented three implementations of the label propagation algorithm for community detection on hypergraphs, targeting different programming models (SYCL, OpenMP target offload, and Kokkos) across three heterogeneous architectures (AMD MI100, Intel Max 1550, and NVIDIA V100S). We introduced a consistent set of optimizations across all models to mitigate load imbalance and

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE

improve memory access efficiency for sparse hypergraphs. Experimental results demonstrated that SYCL consistently achieved the best performance on the largest datasets and the best stability across all hardware platforms. Finally, the results highlighted that SYCL offered the most performance-portable scenario, with a PP metric reaching up to 80%.

ACKNOWLEDGMENT

This work has been partially supported by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU.

REFERENCES

[1] D. Jin, Z. Yu, P. Jiao, S. Pan, D. He, J. Wu, P. S. Yu, and W. Zhang, "A survey of community detection approaches: From statistical modeling to deep learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1149–1170, 2023.

[2] F. Battiston, G. Cencetti, I. Iacopini, V. Latora, M. Lucas, A. Patania, J.-G. Young, and G. Petri, "Networks beyond pairwise interactions: Structure and dynamics," *Physics Reports*, vol. 874, pp. 1–92, 2020.

[3] A. Antelmi, G. Cordasco, M. Polato, V. Scarano, C. Spagnuolo, and D. Yang, "A survey on hypergraph representation learning," *ACM Computing Surveys*, vol. 56, no. 1, 2023.

[4] A. Bretto, "Hypergraph theory," *An introduction. Mathematical Engineering. Cham: Springer*, vol. 1, pp. 209–216, 2013.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[6] TOP500.org. (2025) June 2025 — TOP500. [Online]. Available: https://top500.org/lists/top500/2025/06/

[7] A. Antelmi, G. Cordasco, B. Kamiński, P. Prałat, V. Scarano, C. Spagnuolo, and P. Szufel, "Analyzing, exploring, and visualizing complex networks via hypergraphs using SimpleHypergraphs.jl," *Internet Mathematics*, 2020.

[8] S. E. Garza and S. E. Schaeffer, "Community detection with the label propagation algorithm: A survey," *Physica A: Statistical Mechanics and its Applications*, vol. 534, p. 122058, 2019.

[9] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, vol. 76, no. 3, p. 036106, 2007.

[10] T. K. S. W. Group. (29-03-2023) SYCL 2020 Specification (revision 8) — registry.khronos.org. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[11] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 5.2*, November 2021, https://www.openmp.org/specifications/.

[12] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*, 1994.

[14] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, "Simplicial closure and higher-order link prediction," *Proceedings of the National Academy of Sciences*, vol. 115, no. 48, pp. E11 221–E11 230, 2018. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1800683115

[15] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. P. Hsu, and K. Wang, "An overview of microsoft academic service (mas) and applications," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15 Companion, 2015, p. 243–246. [Online]. Available: https://doi.org/10.1145/2740908.2742839

[16] B. Luo, "eventernote-places," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11263394

[17] N. Landry, "senate-bills," Apr. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10957697

[18] N. W. Landry, M. Lucas, I. Iacopini, G. Petri, A. Schwarze, A. Patania, and L. Torres, "Xgi: A python package for higher-order interaction networks," *Journal of Open Source Software*, vol. 8, no. 85, p. 5162, 2023.

[19] S. J. Pennycook and J. D. Sewall, "Revisiting a metric for performance portability," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 1–9.

[20] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.

[21] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07, 2007, p. 21. [Online]. Available: https://doi.org/10.1109/MSR.2007.31

[22] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang, "Hyperx: A scalable hypergraph framework," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 5, pp. 909–922, 2019.

[23] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20, 2020, p. 232–249.

[24] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, p. 135–146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[25] Q. F. Lotito, M. Contisciani, C. De Bacco, L. Di Gaetano, L. Gallo, A. Montresor, F. Musciotto, N. Ruggeri, and F. Battiston, "Hypergraphx: a library for higher-order network analysis," *Journal of Complex Networks*, vol. 11, no. 3, 05 2023.

[26] N. W. Landry, M. Lucas, I. Iacopini, G. Petri, A. Schwarze, A. Patania, and L. Torres, "XGI: A Python package for higher-order interaction networks," *Journal of Open Source Software*, vol. 8, no. 85, p. 5162, May 2023.

[27] B. Praggastis, S. Aksoy, D. Arendt, M. Bonicillo, C. Joslyn, E. Purvine, M. Shapiro, and J. Y. Yun, "HyperNetX: A Python package for modeling complex network data as hypergraphs," *Journal of Open Source Software*, vol. 9, no. 95, p. 6016, Mar. 2024.

[28] S. Sahu, M. N, and K. Kothapalli, "ν-lpa: Fast gpu-based label propagation algorithm (lpa) for community detection," in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, pp. 395–404.

[29] L. Mišić and B. Žalik, "Parallel implementation of the label propagation method for community detection on the gpu," in *Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 851–856.

[30] W. Ye, X. Liang, Y. Wu, and Y. Wen, "Glp: A gpu-based label propagation framework for large-scale graphs," *Journal of Parallel and Distributed Computing*, 2023.

[31] T. Fiscarelli and M. R. Brust, "Local memory boosts label propagation for community detection," *Applied Network Science*, vol. 4, no. 1, pp. 1–14, 2019.

[32] I. Z. Reguly, "Evaluating the performance portability of sycl across cpus and gpus on bandwidth-bound applications," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1038–1047.

[33] W. L. Lee, D.-L. Lin, C.-H. Chiu, U. Schlichtmann, and T.-W. Huang, "Hyperg: Multilevel gpu-accelerated k-way hypergraph partitioner," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '25, 2025, p. 1031–1040. [Online]. Available: https://doi.org/10.1145/3658617.3697551

Authors preprint.
Not for redistribution.
The definitive version was published at PDP 2026, © IEEE