SYgraph: A Portable Heterogeneous Graph Analytics Framework for GPUs

Antonio De Caro
antdecaro@unisa.it
University of Salerno
Salerno, Italy

Gennaro Cordasco
gcordasco@unisa.it
University of Salerno
Salerno, Italy

Biagio Cosenza
bcosenza@unisa.it
University of Salerno
Salerno, Italy

Abstract

Graph analytics play a crucial role in a wide range of fields, including social network analysis, bioinformatics, and scientific computing, due to their ability to model and explore complex relationships. However, optimizing graph algorithms is inherently difficult due to their memory-bound constraints, often resulting in poor performance on modern massively parallel hardware. In addition, most state-of-the-art implementations are designed in CUDA for NVIDIA GPUs, and thus they can not run on supercomputers equipped with AMD and Intel GPUs. To address these challenges, we propose SYgraph, a portable heterogeneous graph analytics framework written in SYCL. SYgraph provides an efficient two-layer bitmap data layout optimized for GPU memory, eliminates the need for pre- or post-processing steps, and abstracts the complexity of working with diverse target platforms. Experimental results demonstrate that SYgraph delivers competitive performance against state-of-the-art frameworks on datasets with up to 21 million nodes and 530 million edges on NVIDIA GPUs while being able to target any SYCL-supported device, such as AMD and Intel GPUs.

CCS Concepts

• **Computing methodologies → Parallel computing methodologies**; **Graphics processors**.

Keywords

Graph Analytics, GPU, Parallel Computing, Portability, Bitmap Frontier

ACM Reference Format:
Antonio De Caro, Gennaro Cordasco, and Biagio Cosenza. 2025. SYgraph: A Portable Heterogeneous Graph Analytics Framework for GPUs. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3754598.3754615

1 Introduction

Graph processing is essential for examining large, complex networks in fields such as social network analysis, bioinformatics, and cybersecurity. These uses underscore the value of graph analytics to gain insight from interconnected data. Nevertheless, optimizing graph algorithms on modern parallel hardware is challenging due to their memory-bound characteristics, which frequently lead to suboptimal performance.

State-of-the-art graph frameworks like Gunrock [36], Tigr [25], and SEP-Graph [33] are developed in CUDA, making them specific to NVIDIA GPUs. This limits their use on other high-performance GPUs from companies like AMD and Intel. With 7 out of the top 10 supercomputers now featuring GPUs from these vendors [31], there is a crucial need for a graph framework that can run efficiently on diverse hardware platforms.

We propose SYgraph [1], an efficient and portable graph analytics framework using SYCL and C++20 to tackle these challenges. SYCL [17] offers an abstraction layer for single-source programming on heterogeneous systems, enabling SYgraph to seamlessly support multiple GPU backends like CUDA, ROCm, LevelZero and OpenCL. SYgraph allows developers to write graph analytics code that runs efficiently on many GPU architectures with a unified programming style.

SYgraph facilitates developers in efficiently applying graph algorithms via its API. This setup ensures efficient execution of essential graph algorithms like breadth-first search, single-source shortest path, betweenness centrality, and connected components, without requiring deep GPU optimization knowledge, thus broadening its applicability. SYgraph introduces a Two-Layer Bitmap data structure that improves the traditional bitmap method, allowing an integer to represent up to 64 active elements, thereby improving memory efficiency and GPU resource utilization.

This paper discusses SYgraph's design and implementation, focusing on the architectural choices and optimizations that enhance its performance. Through thorough benchmarking and comparison, we show that SYgraph surpasses leading frameworks such as Gunrock[36] up to 2.10×, Tigr[25] up to 4.53×, and SEP-Graph[33] up to 2.07× on a dataset consisting of 21 million nodes and 530 million edges on the NVIDIA V100S GPU. In addition, we emphasize SYgraph's capability to support graph analytics tasks on AMD and Intel GPUs, promoting the growth of heterogeneous GPU systems in this field. Our results also highlight the differing efficiencies of specific GPU architectures across various graph datasets.

In summary, this paper makes the following contributions:

- SYgraph, the first portable, heterogeneous graph analytics GPU framework based on SYCL and C++20, capable to target different GPU architectures;
- A novel approach to graph frontier management, entirely designed around a Two-Layer bitmap representation and integrating a custom load-balancing strategy. This bitmap-centric approach reduces memory footprint, eliminates pre-

[1]Source code and documentation available at *https://github.com/unisa-hpc/SYgraph*

and post-processing overhead, and enables efficient data-driven traversal operations on GPUs;
- A comparative experimental study of SYgraph against established state-of-the-art graph analytics GPU frameworks on BC, BFS, CC, and SSSP algorithms on different datasets, including a performance evaluation across AMD, Intel, and NVIDIA GPUs.

2 Related Work

Graph processing frameworks have evolved to meet the varied needs of graph analytics on different platforms. Numerous research efforts have focused on processing graphs on multi-core systems. Galois [24] leverages amorphous data-parallelism and optimistic parallel execution to efficiently run irregular algorithms. GraphIt [40] uses a domain-specific language and compiler for graph tasks, enabling optimization by separating algorithm definitions from scheduling. Similarly, GraphMat [30] offers a high-level API that turns operations into efficient matrix-vector computations with CPU parallelism. Ligra [28] aims to use parallel graph traversal algorithms for shared memory.

2.1 GPU-Based Frameworks

The large-scale parallel processing capabilities of GPUs have been leveraged by various frameworks to adapt the irregular nature of graph algorithms for GPU use. There are two main approaches to graph processing on GPUs. The *frontier-based* approach involves a frontier reflecting active vertices or edges during computations. Within this group are Gunrock [36], a high-performance library with diverse graph primitives tailored for GPUs, emphasizing user-friendliness and performance through abstract models. Tigr [25] optimizes memory access for balanced workloads and high efficiency. SEP-graph [33] focuses on streaming partitioned edges into GPU memory to enhance bandwidth utilization. Grus [35] provides a versatile platform for GPU-based graph analytics, with support for dynamic graph processing. CuSha [20] introduces G-Shards for improved memory coalescence and divergence reduction. CuGraph [15] is part of the RAPIDS suite, offering graph analytics algorithms optimized for NVIDIA GPUs. GraphBLAST [39] efficiently uses sparse linear algebra on GPUs by converting graph tasks into matrix/vector operations, utilizing high-performance libraries. Although all of these frameworks are tied to CUDA and limited to NVIDIA GPUs, AMD's HIP enables porting to ROCm through source-to-source translation. However, it often requires substantial manual intervention and duplicated code, increasing complexity. As such, HIP is better suited for migration than for true portability.

With respect to other GPU-based frameworks, SYgraph is the first portable framework that offers a unified API, enabling seamless integration across different hardware architectures.

2.2 Frontier Data Layout and Workload Balancing

Gunrock [36] uses a dynamic vector for the frontier, enabling flexible vertex and edge operations during graph traversal. This method suits graph processing's irregular nature, where frontier sizes can change greatly between iterations. To combat workload imbalance, Gunrock uses advance operations to redistribute workloads among threads and mixes push-pull traversal strategies to effectively balance computations. However, it requires post-processing to remove duplicate nodes for frontier consistency and reallocates memory when the vector is full. GraphBLAST [39], though centered on linear algebra, uses a matrix-based layout for the frontier. By translating graph tasks into sparse matrix operations, it gains from linear algebra library optimizations, efficiently managing large frontiers through sparse matrix-vector multiplication. Its workload balance is achieved through the parallel nature of sparse linear algebra. SEP-graph [33] switches between vector and bitmap layouts to remove duplicate nodes and improve execution correctness. To utilize memory bandwidth fully, it dynamically inputs edges into GPU memory, adapting execution modes—synchronous/asynchronous, push/pull, data-/topology-driven—based on workload. This adaptability, however, introduces a runtime overhead sometimes surpassing the algorithm's computational cost. Tigr [25] offers a different approach by directly traversing the graph, avoiding the typical frontier model. This reduces complexity but limits flexibility, needing algorithms specially designed for GPU parallelism. Tigr reduces imbalance by using Uniform-Degree Tree Transformations (UDT) to split high-degree nodes into smaller, uniform structures.

In contrast to those works, SYgraph introduces a novel frontier representation using a bitmap layout optimized for GPUs, alongside a workload balancing strategy designed specifically for this data layout, resulting in enhanced memory efficiency while eliminating the need for post-processing tasks.

	SYgraph	Gunrock [36]	Tigr [25]	SEP-Graph [33]
Targeted Arch.	Heterogeneous	CUDA	CUDA	CUDA
Pre-Processing	No	No	Yes	Yes
Post-Processing	No	Yes	Yes	Yes
Data-Layout	Two-Layer Bitmap	Vector	Adj. List	Vector/Bitmap
Execution Model	Sync	Sync	Sync	Sync/Async
Load Balancing	Bitmap-tailored	Dynamic task re-distribution	Node riorganiza-tion	Algorithmic

Table 1: Comparison against the state of the art.

Table 1 provides a comparison table of SYgraph with some of the state-of-the-art graph analytics frameworks on GPU. Heterogeneous means that it supports all SYCL-enabled GPU back-ends.

3 SYgraph Overview

SYgraph is a high-performance graph processing framework that leverages GPU computational power. It's a C++ header-only library that offers primitives for manipulating and analyzing large-scale graphs efficiently.

As shown in Figure 1, the SYgraph framework is structured into three layers: the application layer providing the API for developers, the SYgraph core layer containing the framework's main implementation, and the SYCL layer ensuring platform portability.

The SYgraph framework manages a frontier, which is the set of active vertices or edges during a graph algorithm iteration. According to Wang et al. [36], this approach allows the practical implementation of complex graph algorithms on a GPU. Our framework follows the Bulk Synchronous Parallel (BSP) model [16], executing

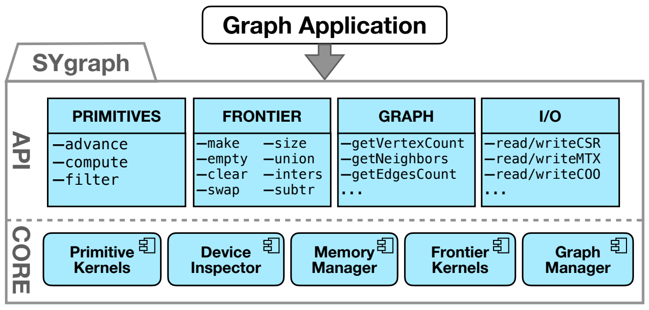


Figure 1: SYgraph architecture.

Function	Description
`namespace sygraph::operators::`	
`advance::vertices(Graph, OutFrontier, Functor)`	Traverses outgoing edges of all vertices using `Functor`, storing results in `OutFrontier`.
`advance::vertices(Graph, Functor)`	Same as above, without storing the result.
`advance::frontier(Graph, InFrontier, OutFrontier, Functor)`	Traverses outgoing edges from `InFrontier` using `Functor`, storing results in `OutFrontier`.
`advance::frontier(Graph, InFrontier, Functor)`	Same as above, without storing the result.
`filter::inplace(Graph, Frontier, Functor)`	Removes elements from `Frontier` that fail `Functor`.
`filter::external(Graph, InFrontier, OutFrontier, Functor)`	Copies `InFrontier` elements passing `Functor` into `OutFrontier`.
`compute::execute(Graph, Frontier, Functor)`	Applies `Functor` to elements in `Frontier`.
Functors description	
Advance `Functor(src, dst, edge_id, weight) -> Bool`	Determines whether the `dst` vertex should be added to the frontier.
Filter `Functor(id) -> Bool`	Returns whether a vertex `id` should remain in the frontier.
Compute `Functor(id)`	Operates on a given `vertex_id`.

Table 2: SYgraph API Overview of Primitives.

one of the SYgraph core primitives in each superstep. At the end of each superstep, the output frontier is updated according to the computations, maintaining consistency across GPU threads. Although supersteps may have interdependencies, operations within each step run independently, simplifying integration with existing applications without needing to address operation synchronization.

An important design principle of SYgraph is enabling data-driven operations through user-defined C++ lambda functions for each superstep's computations. These lambdas integrate with GPU kernel primitives to manage the computational state and update the output frontier. In BFS, for instance, a lambda checks if a vertex is visited, ensuring only new vertices are added to the output frontier.

3.1 SYgraph API

The SYgraph API consists of four main components: Primitives, Frontier, Graph, and Input/Output (IO) API. While the IO API defines a set of functions for reading and writing graphs from and to files, in this section, we will focus on the other APIs. Table 2 summarizes the primitives methods in the SYgraph API.

Primitives. The SYgraph framework, like Gunrock [36], has three main primitives for data-driven graph manipulation using a lambda function that interacts with user data structures. These are *advance*, *compute*, and *filter*. While typically synchronized, some operations can run asynchronously, such as two advance functions on separate graphs. Each primitive returns an event for host-side waits. The **advance** primitive is key for graph applications by determining vertex neighborhoods in active frontiers, requiring load-balancing strategies for handling irregular graph structures. It can also be applied to all vertices, such as in initializing Betweenness Centrality. The **compute** primitive updates the values of the frontier elements and is kept separate from the advance because it does not present the same load balancing challenges. The **filter** primitive removes elements from a frontier based on criteria, either in place or by creating a new frontier.

Frontier. A SYgraph Frontier represents the collection of active elements in graph computations. Developers can employ a *frontier* object for graph primitives as input and output parameters, determine its status (e.g., count of active elements), and add or remove elements. Graph algorithms frequently execute operations on the currently visited vertex set. In such cases, frontier operations aid in efficiently managing the active vertices during the algorithm's execution. Thus, we offer a suite of fundamental frontier API tools for these tasks. The **intersection** operator identifies the shared neighborhood of two active node sets. The **union** operator joins two active node sets into one frontier (e.g., in graph machine learning [37]). **Subtraction** removes specific nodes from a set for focused analysis or computation (e.g., data cleaning).

Graphs Representations. SYgraph primarily offers CSR and CSC graph representations [4]. However, the SYgraph API lets users define their own graph representations by implementing an interface containing the necessary methods and structs for the SYgraph primitives. Users also need to create an *iterator* class for vertex neighbor iteration. Flexibility in graph representation is essential because modern frameworks improve graph analytics on GPUs using GPU-optimized graph structures or variations of CSR and CSC [25]. This flexibility is vital to meet the varied demands of different applications and datasets. Moreover, user-defined custom graph representations can improve performance and scalability in dynamic graphs, which require efficient data structures and algorithms for GPU processing as they evolve with vertex or edge changes [3, 12].

3.2 SYgraph Core

The SYgraph Core includes various components that together execute the API functions. The *primitive kernels* establish the main framework for the primitive functions, incorporating user-defined lambdas. Meanwhile, the *frontier kernels* provide key operations for handling frontiers, including applying frontier operators such as intersection, union, and subtraction, as well as retrieving frontier size, verifying if a frontier is empty, and managing frontiers by clearing and swapping. The *graph manager* performs the usual graph tasks such as obtaining the neighborhood of a vertex, computing the degrees of the vertex, and other graph-related functions. Furthermore, the *device inspector* assesses the target GPU on the fly to fine-tune parameters like thread block size, coarsening factor, and memory layout. Lastly, the *memory manager* manages GPU memory allocation and deallocation for components such as the frontier, graph, and temporary data structures.

3.3 SYgraph Backend

Utilizing SYCL, SYgraph easily integrates CUDA, ROCm, OpenCL, and LevelZero backends. A `queue` in SYCL is used for submitting kernels and transferring data with its linked device. Developers must specify the queue before allocating a graph or frontier object to select the offloading device.

In SYgraph, Graph and Frontier objects are created using SYCL Unified Shared Memory (USM) with the `malloc_shared` allocator. USM offers a unified memory model that facilitates automatic data transfers between the host and the device. Additionally, USM supports setups where GPU and CPU share address space. SYCL's abstraction layer enhances portability, but introduces some overhead, affecting performance, especially on specific hardware. On AMD hardware, USM is activated by Xnack [1], where we noticed suboptimal performance. To address this, developers can choose between USM and explicit memory allocation at compile time.

SYgraph implements primitives using SYCL's `parallel_for`, which assigns tasks to each GPU thread. The *advance* primitive uses an `nd_range` for detailed control over global and local work sizes, facilitating balanced workload and optimized memory access. Conversely, *compute* and *filter* use a `range`, which specifies only a global range, leaving the thread block division to the SYCL compiler.

3.4 Algorithm Implementations

A graph algorithm can be seen as an iterative process that converges as vertex or edge attributes are updated. Convergence is reached when further iterations cause no major changes, signaling algorithm stability. This applies to basic algorithms such as *Betweeness Centrality* (BC), *Breadth First Search* (BFS), *Connected Components Labeling* (CC), and *Single Source Shortest Path* (SSSP), where convergence occurs once all nodes are visited and no more updates are needed, achieved by exploring the graph from a starting point. Implementing graph algorithms with SYgraph involves focusing on convergence.

Taking advantage of Brandes' formulation [9], the BC implementation computes the number of edges through each vertex by traversing the graph first forward, then backward, from a source vertex. Much like Gunrock [36], BFS begins at one vertex and creates new frontiers each iteration via advance operations. Convergence occurs once all vertices are visited. Our BFS uses the push-based method, but it is also possible to use both push and pull techniques as per Beamer et al. [5]. The CC algorithm follows a label propagation method as outlined by Stergiou et al. [29], where vertices begin by distributing their labels to neighbors. The process stops when no label changes occur. The SSSP algorithm employs Bellman-Ford [11] to calculate the shortest paths from a source vertex to all others by minimizing path weights. The advance phase resembles the BFS, moving from one vertex to adjacent ones and updating distance values. Our SSSP version does not use the Δ-stepping optimization discussed in [23, 34].

```
1  using namespace sygraph;
2  void BFS(Graph& G, size_t* dist, vertex_t src){
3  auto in_frontier  = makeFrontier<frontier_view_t::vertex>(G);
4  auto out_frontier = makeFrontier<frontier_view_t::vertex>(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 }}
```

Listing 1: BFS in SYgraph. Keywords specific to SYgraph within the `sygraph` namespace are displayed in purple.

3.5 SYgraph in Action

Listing 1 outlines the main elements of the framework, such as the frontier-based execution model and the operators used to update and calculate graph states. The highlighted lines run on the GPU. Line 2's `Graph` object is linked to a queue and specific device. The `distance` and `parents` vectors are allocated using `malloc_device`. Lines 3-4 set up the input and output frontiers in a vertex view, indicating the vertices being processed in the current and upcoming iterations.

The advance operator (lines 9-13) expands the current frontier by exploring neighboring vertices, using a user-defined lambda function to decide which vertices to visit and avoid re-exploration. Once the frontier is updated, the compute operator (lines 14-17) updates vertex properties such as the distance from the source by applying a lambda function to all vertices in the output frontier, with calculations parallelized across GPU threads. Although the compute operator can be combined with the advance operator, it does not face load-balancing issues, leading to efficient global memory access and improved performance. However, it requires waiting for an additional GPU kernel to complete. The framework iterates through the graph smoothly by swapping frontiers at each step's end (line 18), and the clear function (line 19) prepares the output frontier for the next cycle. This process repeats until the input frontier is empty (line 8). In this scenario, we assumed that the distance vector is initialized outside of the function.

4 Frontier Data Layout

In graph traversal operations, vectors-based frontiers add discovered vertices to a vector. A common technique [21] employs the GPU's local shared memory to efficiently synchronize the vector's end. This involves a small local array that allows for more efficient atomic operations for synchronization at the vector tail than when using global memory. When the local array fills up, synchronization shifts to the global tail. A prefix sum is then computed among local tails of *thread blocks*, and the data from the local array is coalesced into the global memory.

One criticism of this approach is that due to the irregular nature of graphs, an uneven number of vertices may be processed on different GPU *thread blocks*, causing only a few blocks to fully utilize

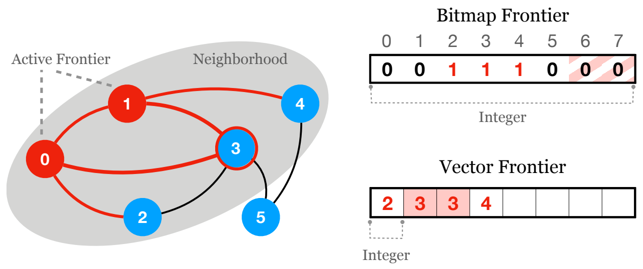


Figure 2: Bitmap frontier vs vector-based frontier during a BFS iteration.

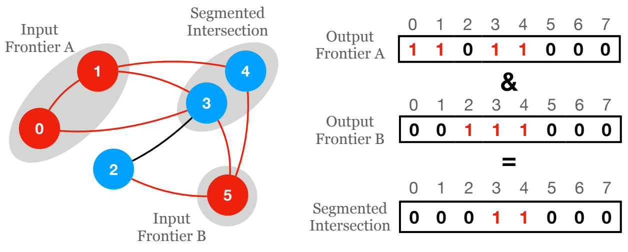


Figure 3: Segmented Intersection computed by leveraging the bitmap representation.

local memory before synchronizing with global memory. Another key concern is the vector-based frontier's issue with duplicate elements. To address this, state-of-the-art frameworks transition from vector to bitmap representation between steps [14, 28, 33], or scan vectors to remove duplicates [6, 15, 36]. Furthermore, vector-based frontiers can be inefficient in space, particularly with high-degree vertices typical of social network graphs, where the number of active vertices in a frontier rapidly increases in early iterations. Meng et al. [22] delve deeper into these issues. Therefore, we focused on creating a memory-efficient data layout, like bitmaps, to represent active vertices in each algorithm step.

4.1 Bitmap Implementation

Our approach utilizes a bitmap format, which provides multiple benefits over traditional vector-based methods. By representing each vertex or edge's active state with just one bit, memory usage is greatly minimized, especially beneficial for high-degree vertex graphs where vectors demand more space. Additionally, synchronization is simplified as atomic operations set bits in the bitmap to represent active vertices, avoiding the complexity of vector management and prefix sum calculations.

The bitmap is composed of an array of unsigned integers (either 32 or 64 bits), each linked to a specific set of graph vertices. Thus, each vertex corresponds to one bit within a particular integer. The bitmap size for a graph $G = (V, E)$ is given by $\lceil |V|/b \rceil$, where b represents the bit size of each integer. Figure 2 illustrates the contrast between bitmap- and vector-based frontiers, showing that a bitmap saves more memory by using a single 8-bit integer for vertex representation, and also prevents vertex duplication, as demonstrated with the vertex labeled 3.

In an advance operation, when vertex u detects vertex v, v is added to the frontier in two steps: (1) locate the bitmap array index through $id(v)/b$, where $id(v)$ is v's ID and b is the number of bits per array element; (2) find the corresponding bit using $id(v)$ mod b. This method assigns each vertex a unique bit, naturally preventing duplicates and eliminating extra post-processing. The Grus framework [35] opted for a *boolmap* method, linking each vertex to a byte, but this increases memory use eightfold.

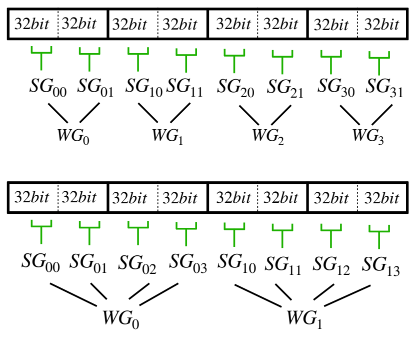
Frontiers Operators. By portraying the frontier as a bitmap, the **intersection**, **union**, and **subtraction** operations are efficiently executed. These frontier transformations can run parallelly via bitwise operations: intersection through bitwise AND, union via bitwise OR, and symmetric difference using bitwise XOR. This method takes advantage of parallelism by mapping each integer in the bitmap to a GPU thread, optimizing frontier transformations. Figure 3 illustrates the segmented reduction (finding the common neighborhood of two frontiers) using the **intersection** operator.

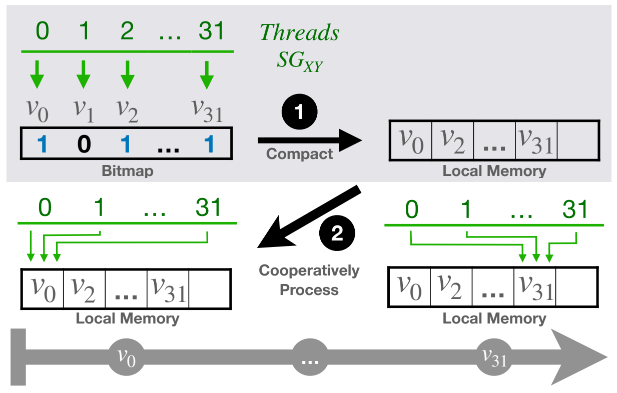
4.2 Workload Mapping

In this section, we adopt SYCL naming conventions for the GPU platform model. A *workitem* (T) describes a GPU thread mapped to one kernel execution. A *workgroup* (WG) is a one-, two-, or three-dimensional thread set, similar to a *block* in CUDA. A *subgroup* (SG) indicates a range of consecutive workitems processed as a SIMT vector, called a *warp* in CUDA and a *wavefront* in AMD. With respect to the memory model, *local memory* in SYCL corresponds to *shared memory* in CUDA—a user-managed memory region accessible by all workitems within the same workgroup.

The workload balancing on the bitmap involves multiple threads assisting each other in processing one vertex's neighborhood at a time. In SYgraph, this is known as *workgroup-mapped* load balancing, where each workgroup handles an integer of the bitmap. To optimize GPU use, a coarsening factor is introduced, defining how many integers a workgroup processes, as illustrated in Figure 4a. For a 64-bit integer: with a coarsening factor of 1, a workgroup handles 64 vertices; a factor of 2 means 128 vertices. This method achieves efficient load balancing by leveraging the GPU's intra-subgroup capabilities for even thread workload distribution. Each subgroup handles a certain part of the bitmap integer, and the size of this part depends on both bitmap and subgroup size. Whereas NVIDIA and AMD GPUs have fixed subgroup sizes, Intel GPUs allow flexibility with sizes of 16 or 32 threads in SIMT on Intel MAX 1100. This adaptability improves performance by allowing better use of GPU resources.

Intra-subgroup processing involves two stages, as shown in Figure 4b. Initially ❶, subgroup collectives, such as scan operations, are used to compact active vertices into local memory. The local memory for each workgroup is defined by the coarsening factor and the range of a bitmap's single integer. Next ❷, each workitem in the subgroup cooperatively handles a distinct region of the neighborhood for each vertex previously placed into local memory. This method avoids the need for synchronization, allowing threads in the same subgroup to process separate neighbors independently without conflicts (Figure 4c).


(a) Work-group mapping with coarsening factor 1 above, and 2 below. The bitmap frontier is represented by 64bit integers.


(b) Intra-subgroup load balancing.

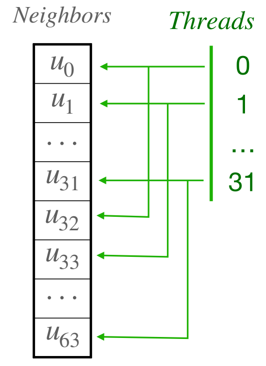
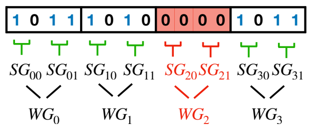

(c) Cooperative neighborhood processing.

Figure 4: Load balancing of the advance operation.


(a) A work-group is associated to a 0-value bitmap integer.

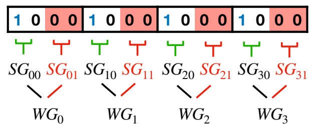

(b) Only one of the sub-groups is active.

Figure 5: Worst case scenarios on bitmap-based frontier.

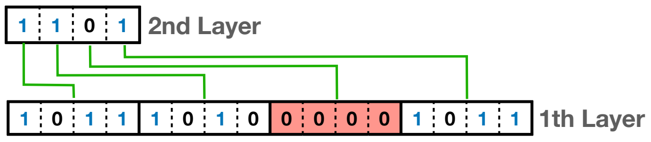


Figure 6: Example of 2LB data structure. For illustration purposes, the bitmap is represented by 4-bits integers.

Access to global memory for neighborhood processing can be coalesced, improving memory efficiency, depending on the graph representation (e.g., CSR or CSC). If a user employs a custom graph representation, they must ensure neighborhood data locality.

4.3 Two-Layer Bitmap Data Layout

In developing this bitmap approach, we faced two major edge cases affecting performance: (1) all bits of a bitmap integer are 0, wasting the workgroup's processing capacity (Figure 5a), and (2) only one bit is 1, activating just one subgroup (Figure 5b).

First, consider scenario (2). To keep only one subgroup active, utilize a 32-bit integer to match a warp on NVIDIA GPUs and a 64-bit integer for AMD GPUs to align with the wavefront size. For Intel GPUs, set the bitmap integer to 32 and select a subgroup size of 32 threads. Furthermore, *adjust the coarsening factor to keep the entire compute unit active*, enhancing GPU resource usage.

To conserve group resources, we avoid allocating workgroups to integers that are zero. We accomplish this with a **Two-Layer bitmap (2LB)** data layout, which maps primary bitmap integers to a secondary bitmap of size $\lceil |V|/b^2 \rceil$. In this layout, a secondary layer bit is set to 1 if its corresponding primary integer has any bit set to 1, as shown in Figure 6. When adding a vertex, the corresponding bit in the second layer is calculated and set to 1 if it's not already. For vertex removal, if the integer becomes 0, the second layer bit is reset to 0. The second layer behaves similarly to the bitmap discussed in Section 4.1.

Before each *advance* operation, GPU threads map to integers in the second layer to find nonzero integers in the first bitmap layer and store their offsets in a global buffer. During the advance, a set number of workgroups run on the GPU, iterating over the offsets buffer to retrieve integers and apply the load-balancing strategy.

4.4 Comparing 2LB with Existing Solutions

CPU graph frameworks have examined using bitmaps to represent active elements [5, 30, 40]. Bitmaps suit GPU use due to parallel processing. Yet, GPUs need different memory access and optimizations to use bitmaps well. Though memory-efficient, bitmaps can cause issues on GPUs like scattered memory access, thread divergence, and load imbalance, hurting performance. Some frameworks use coarsened bitmaps or warp-wide processing to boost parallelism and cut memory divergence [2, 32]. SYgraph, with its 2LB frontier layout, dedicates GPU threads to nonzero bitmap elements, focusing on relevant areas of the first layer. This enhances spatial locality, cuts memory access, and balances thread workloads, minimizing underutilization and divergence.

Our approach is naturally duplicate-free and introduces new load-balancing techniques and optimizations specifically for bitmap-based GPU processing, unlike other frameworks [33, 35] requiring bitmap-to-queue transitions to remove redundant nodes.

Incorporating extra bitmap layers can refine our 2LB, turning the layout into a bitmap-tree. Research [38] examines bitmap-tree structures on CPUs. Yet, tree-based layouts face difficulties on GPUs due to irregular memory access patterns and warp divergence from hierarchical traversals [10]. Additionally, more than two layers add substantial overhead because of increased computation for nonzero integer offsets and extra synchronization during advance operations. A dynamic number of bitmap levels adds complexity. With fixed levels, the compiler unrolls loops for setting bits to 1; this

Graph	Vertices	Edges	Avg. Deg.	Max Deg.
roadNet-CA (CA)	2M	2,8M	2,8	12
road-USA (USA)	23,9M	28,9M	2,4	9
Hollywood-2009 (hollyw)	1,1M	56,9M	103,4	11K
Indochina-2004 (indo)	7,4M	194,1M	52,4	256K
LiveJournal (journal)	4,8M	69M	28,7	2K
kron-g500-logn21 (kron)	2,1M	91M	86,6	213K
soc-twitter-2010 (twitter)	21,3M	530M	24,8	698K

Table 3: Datasets used in this work, taken from Network Repository [27] and WebGraph [7, 8].

Mach.	Vendor	GPU	VRAM	SYCL Back-End	L2 Cache
A	NVIDIA	Tesla V100S	32GB	CUDA v12.3	6MB
B	Intel	MAX1100	48GB	LevelZero, OpenCL	108 MB
C	AMD	MI100	32GB	ROCm v7.0.0	8 MB

Table 4: Hardware setup of the different architectures employed in the experiments.

optimization is not feasible with dynamic levels since the compiler cannot unroll the loop. To address this, we use SYCL's *specialization constants* to inject dynamic host variables as constants in JIT kernel compilation, although this is efficiently supported mainly on Intel GPUs. In our tests, two layers were used to optimize workload balance and overhead effectively.

5 Experimental Evaluation

We compared SYgraph against three state-of-the-art GPU frameworks: Gunrock [36], Tigr [25], and SEP-Graph [33]. Furthermore, we evaluated the performance of SYgraph on three different hardware configurations described in Table 4. All execution times reported exclude the time required to transfer the graph from the host to the device memory. For compiling SYgraph, we utilized *oneAPI* SYCL compiler v2024.2.1 provided by Intel[18], and CUDA v12.3.

The experimental datasets consist of six graphs, representing both real-world and synthetic scenarios. Among the real-world datasets are *soc-Twitter-2010 (twitter)* and *Hollywood-2009 (hollyw)*, which are social network graphs, as well as *Indochina-2004 (indo)*, a web hyperlink directed graph gathered from Indochina domain sites. The synthetic dataset *kron-g500-logn21 (kron)* exemplifies a graph generated using the R-MAT model [13]. These four datasets are characterized by scale-free properties, with diameters below 20 and highly skewed node degree distributions. In contrast, the *roadNet-CA (CA)* and *road-USA (USA)* datasets are distinguished by their large diameters and more uniform node degree distributions, with most nodes having degrees of 12 or less. These datasets are summarized in Table 3.

5.1 Bitmap Optimizations

As illustrated in Figure 7, the optimizations of Section 4.3 offer significant benefits. Tests were conducted on the *Indochina-2004* dataset by running BFS from a common source, comparing these optimizations to simple bitmap usage. By enhancing GPU resource

	CA		USA		hollyw		indo		twitter		kron	
	L1H	Occ	L1H	Occ	L1H	Occ	L1H	Occ	L1H	Occ	L1H	Occ
SYgraph	**87%**	88%	**87%**	**93**%	**87%**	**92**%	**87%**	90%	**92%**	89%	**91%**	90%
Gunrock	26%	89%	8%	88%	13%	91%	32%	91%	15%	**91**%	4%	88%
Tigr	11%	86%	16%	84%	56%	87%	54%	87%	48%	87%	25%	**93**%
SEP	75%	**90**%	66%	90%	76%	87%	78%	**92**%	77%	**91**%	51%	90%

Table 5: Peak of hardware metrics (i.e. L1 Hit-Rate and Occupancy), during BFS on each dataset on the V100S.

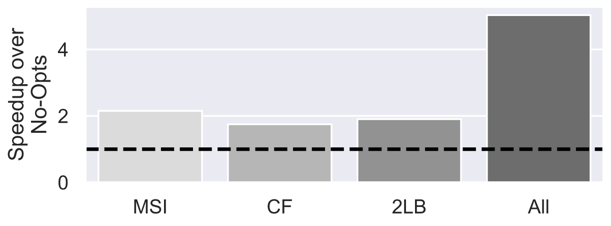


Figure 7: Speedup of optimizations on NVIDIA V100S. *MSI* matches subgroup size to bitmap integer size; *CF* applies coarsening to maximize GPU utilization; *2LB* uses a Two-Layer Bitmap layout; *All* combines all optimizations.

use with the coarsening factor and applying the novel 2LB data layout to avoid subgroup and workgroup stalls, we achieved a 4.43× speedup. Notably, a bitmap allows a single integer to represent 64 active graph vertices, optimizing cache efficiency over other frameworks. This is especially evident after an advance, as the bits for active vertices are prefetched. Table 5 presents peak L1 cache usage and GPU occupancy during advance steps, providing a measure for GPU thread workload balance, with metrics gathered by NVIDIA's NCU tool [26].

5.2 Comparison Against other Frameworks

The experiments involved uniformly randomly selecting 200 sources for each graph to run the BC, BFS, and SSSP algorithms, while for CC, which doesn't need a source, the experiments were repeated 200 times. Figure 8 on Mach. A (Table 4) displays the comparison, showing the median and standard deviation, and Table 6 shows SYgraph's speedups over other frameworks. Figure 9 shows the GPU memory transferred in kilobytes during BFS execution across three datasets: *roadNet-CA*, *Hollywood-2009*, and *Indochina-2004*, chosen for their varying frontier evolution properties. It's important to consider the SYCL runtime's extra load (part of the experiments) absent in native CUDA frameworks when evaluating SYgraph's performance, yet SYgraph performs competitively in algorithms and datasets, with better memory efficiency.

Versus Gunrock. Gunrock reduces the active frontier by removing redundant nodes after each advance operation. When focusing on scale-free graphs, SYgraph surpasses Gunrock mainly because SYgraph does not need to eliminate duplicates. A significant performance improvement is seen on the *Kron* graph, which—although smaller than *Indochina*—as higher connectivity, leading to many

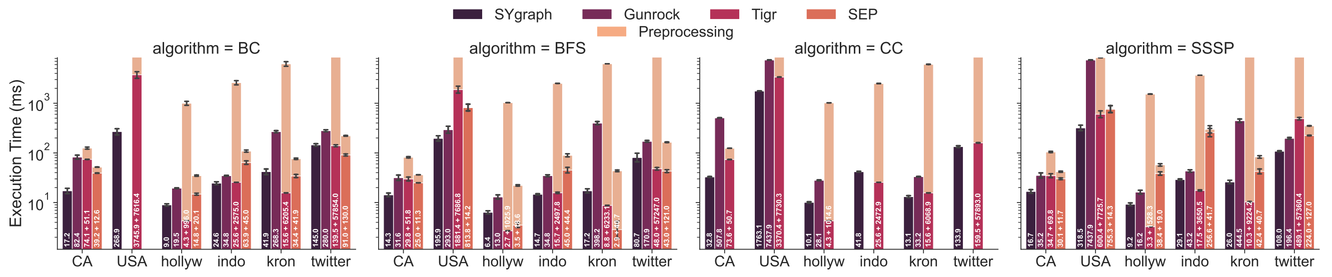


Figure 8: Comparison of SYgraph against the state-of-the-art graph analytics frameworks on NVIDIA V100S GPU. Bar labels show time as $x + y$ (algorithm + preprocessing). When missing, the preprocessing is 0.

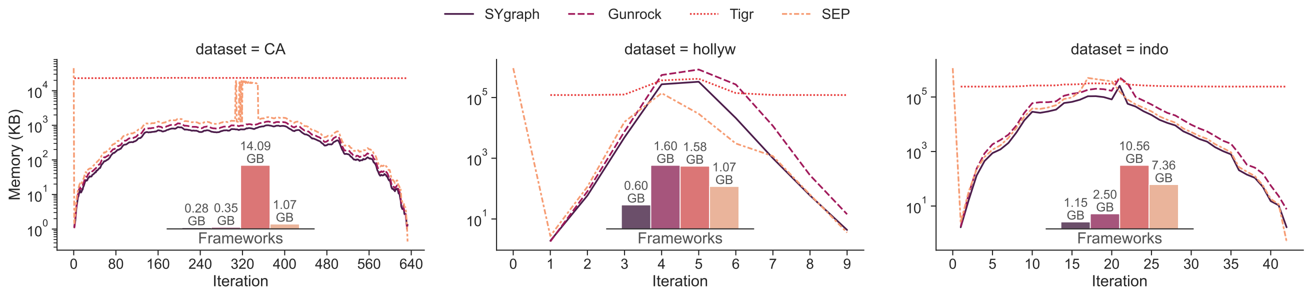


Figure 9: Memory consumption on NVIDIA V100S GPU of SYgraph, Gunrock, Tigr, and SEP-Graph during BFS on three datasets. Lower values indicate greater memory efficiency. In the inset bar charts, each plot represents the total memory consumption of each framework, with the bar sequence matching the order of frameworks listed in the legend.

		CA		USA		hollyw		indo		kron		twitter	
		WPP	WOP	WPP	WOP	WPP	WOP	WPP	WOP	WPP	WOP	WPP	WOP
Gunrock	BC	**4.78**	**4.78**	OOM	OOM	**2.17**	**2.17**	**1.41**	**1.41**	**6.40**	**6.40**	**1.93**	**1.93**
	BFS	**2.22**	**2.22**	**1.50**	**1.50**	**2.04**	**2.04**	**2.37**	**2.37**	**23.12**	**23.12**	**2.10**	**2.10**
	CC	**15.48**	**15.48**	**4.22**	**4.22**	**2.78**	**2.78**	OOM	OOM	**2.53**	**2.53**	OOM	OOM
	SSSP	**2.10**	**2.10**	**23.35**	**23.35**	**1.75**	**1.75**	**1.49**	**1.49**	**17.11**	**17.11**	**1.82**	**1.82**
SEP	BC	**3.00**	**2.27**	OOM	OOM	**3.88**	**1.64**	**4.42**	**2.60**	**1.82**	0.82	**1.52**	0.63
	BFS	**2.54**	**1.75**	**4.23**	**4.15**	**3.48**	0.55	**6.09**	**3.07**	**2.53**	0.17	**2.01**	0.53
	CC	-	-	-	-	-	-	-	-	-	-	-	-
	SSSP	**2.50**	**1.80**	**2.42**	**2.37**	**6.21**	**4.16**	**10.26**	**8.83**	**3.20**	**1.63**	**3.25**	**2.07**
Tigr	BC	**7.26**	**4.30**	**42.25**	**13.93**	>99	0.48	>99	**1.04**	>99	0.37	>99	0.96
	BFS	**5.72**	**2.09**	**48.84**	**9.60**	>99	0.42	>99	**1.07**	>99	0.51	>99	0.59
	CC	**3.79**	**2.24**	**6.30**	**1.91**	>99	0.42	**59.78**	0.61	>99	**1.19**	>99	**1.19**
	SSSP	**6.25**	**2.08**	**26.14**	**1.88**	>99	0.35	>99	0.60	>99	0.40	>99	**4.53**

Table 6: Speedup of the execution time median of SYgraph compared to the other frameworks. WPP includes the preprocessing time, whereas WOP excludes it. OOM stands for an out-of-memory error in the specified framework.

duplicated vertices at each advance step. For the *Indochina* dataset, Gunrock's CC algorithm exhausts memory, but SYgraph maintains high performance due to the 2LB data layout. SYgraph also surpasses Gunrock in memory efficiency. The *Hollywood* and *Indochina* datasets greatly increase Gunrock's memory usage. In contrast, SYgraph's compact data layout minimizes memory usage, especially with numerous high-degree vertices. For road network graphs, SYgraph is more memory-efficient than Gunrock. Furthermore, SYgraph outperforms Gunrock across all algorithms on both the *roadNet-CA* and *road-USA* datasets, with Gunrock running out of memory in the BC algorithm and the USA dataset.

Versus Tigr. Tigr provides GPU-hardwired algorithms and a GPU-friendly CSR variant, but this increases execution time due to graph transformation time. In both the *roadNet-CA* and *road-USA* datasets, SYgraph outperforms Tigr across all algorithms, attributed in part to Tigr's high memory usage for data structure maintenance. As shown in Figure 9, Tigr consumes 14.09 GB for CA, while SYgraph only needs 280 MB. For the *Indochina* dataset, Tigr uses 10 GB compared to SYgraph's 1.15 GB, indicating Tigr's inefficiency with large, sparse road-like graphs. In terms of scale-free graphs, SYgraph outperforms Tigr in all scenarios when considering the preprocessing time. Without considering preprocessing, SYgraph outperforms Tigr in the *Indochina* dataset for BC and BFS, and in *twitter* and *synthetic kron* graphs for the CC algorithm.

Versus SEP-Graph. SEP-Graph stands out from other frameworks by using a hybrid of synchronous and asynchronous execution models, dynamically selecting between push- and pull-based approaches. To prevent node duplication, SEP-Graph converts the queue frontier to a bitmap frontier and then copies values back. We observe shorter preprocessing times compared to Tigr. SYgraph performs well in both scenarios, regardless of whether or not preprocessing time is considered. For the CC algorithm, we couldn't find any implementation compatible with SEP-Graph. SYgraph is more memory efficient than SEP-Graph for scale-free and road-like graphs. Figure 9 shows an initial increase in SEP-Graph's memory

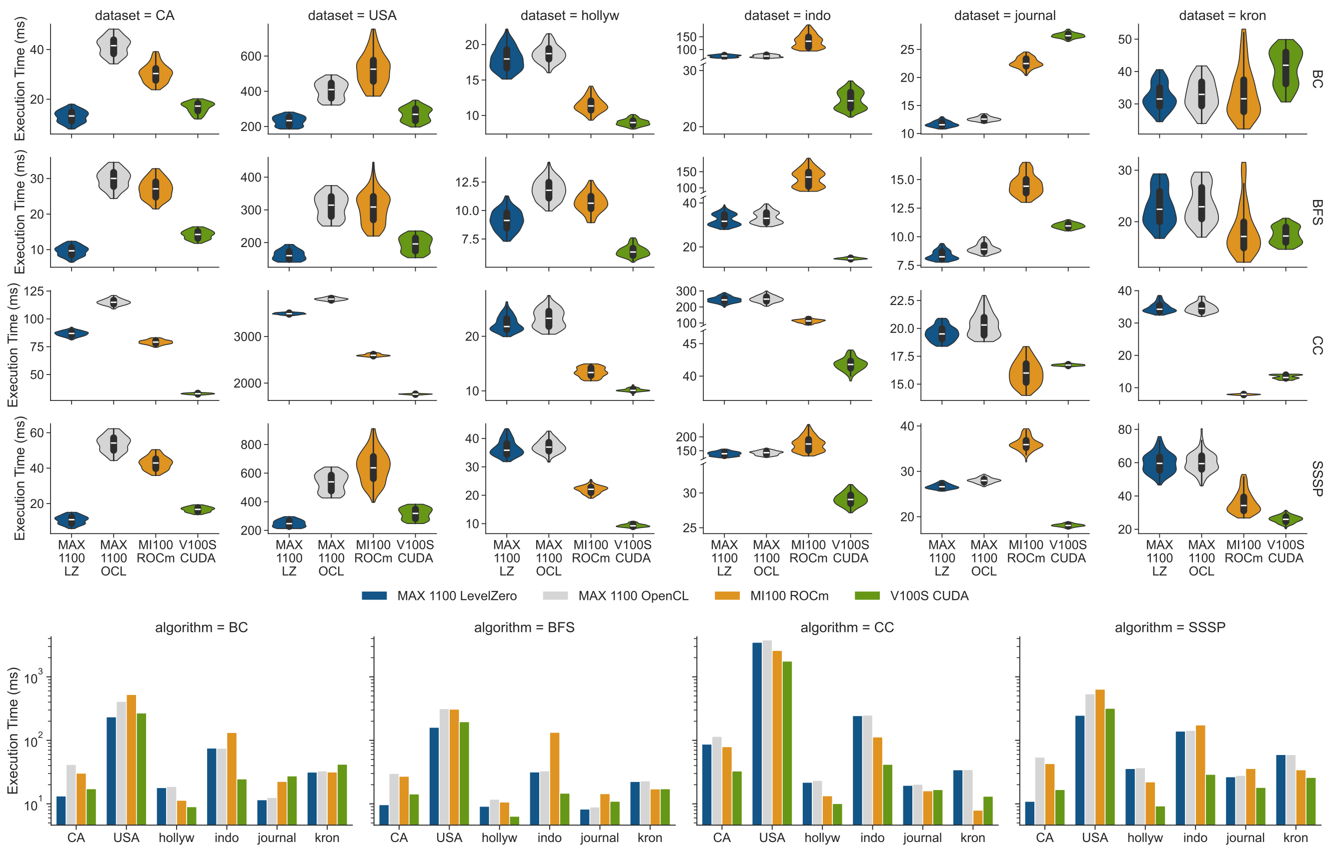


Figure 10: Performance comparison of SYgraph across GPU architectures and backends. Top: distribution per algorithm (rows) and dataset (columns). Bottom: medians on a shared scale for direct comparison.

usage during initialization, which reduces after the first step. In the CA dataset, SEP-Graph hits a mid-computation memory high, due to loading graph data from global memory. We believe that the memory spike in the middle of the computation of SEP-Graph is caused by the switch from push-advance to pull-advance, resulting in more work-items fetching their next edge.

Summarizing, in all datasets and for all algorithms, the geometric mean of speedups shows that SYgraph surpasses Gunrock by 3.49×, Tigr by 7.51×, and SEP-Graph by 2.29× with and without preprocessing.

5.3 Performance Evaluation on Different GPUs

Figure 10 shows SYgraph's performance across the hardware setups detailed in Table 4. These findings show how different hardware architectures and backend APIs affect performance with varying dataset and workload structures. Connected components (CC) inherently involve more intensive computation than traversal-based algorithms like BFS, which helps explain the superior performance of GPUs such as the NVIDIA V100S and AMD MI100 in dense graph scenarios.

The NVIDIA V100S GPU consistently performs well across various datasets and shows particular strength in dense and medium-sparse graphs like *Hollywood*, *Indochina*, and *roadNet-CA*. Its CUDA architecture, with high warp occupancy and efficient memory handling, is particularly effective for compute-heavy workloads. However, in sparse graphs, its performance diminishes, where memory access patterns become more irregular and dominate the execution time, allowing the Intel MAX 1100 to perform comparatively better.

The AMD MI100 GPU, leveraging the ROCm backend, also demonstrates strong performance on dense graphs, surpassing the NVIDIA V100S on datasets such as *LiveJournal* and synthetic *Kron* graphs for CC tasks. In contrast, its performance drops in sparse workloads, where irregular memory access and lower computational intensity limit its efficiency.

6 Conclusion

SYgraph was developed to address the lack of GPU-based graph frameworks that support platforms beyond NVIDIA. It offers a portable, high-performance solution for graph analytics on AMD, Intel, and NVIDIA GPUs, abstracting hardware complexities for users. SYgraph outperforms leading GPU-focused frameworks, exceeding Gunrock by 3.49×, Tigr by 7.51×, and SEP-Graph by 2.29×. The Intel MAX 1100 excels on sparse graphs (e.g., *roadNet-CA*, *road USA*), while the AMD MI100 is better for dense datasets. The NVIDIA V100S offers strong overall performance.

Although currently limited to single-GPU systems, SYgraph is well-suited for multi-GPU and multi-node extensions using static graph partitioning [19], where each GPU handles a local subgraph and can precompute frontier sizes. Dynamic partitioning introduces challenges like frontier reallocation and memory reclamation. Future work will focus on extending SYgraph to support multi-GPU execution.

References

[1] AMD. 2024. *GPU Memory; ROCm Documentation — rocm.docs.amd.com.* https://rocm.docs.amd.com/en/latest/conceptual/gpu-memory.html

[2] Witold Andrzejewski and Robert Wrembel. 2010. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *International Conference on Database and Expert Systems Applications*. Springer, 315–329.

[3] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. 2020. Dynamic Graphs on the GPU. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

[4] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM.

[5] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. https://doi.org/10.1109/SC.2012.50

[6] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/3018743.3018756

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 587–596.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[9] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[10] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[11] Federico Busato and Nicola Bombieri. 2015. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2222–2233.

[12] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547541

[13] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[14] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[15] Alex Fender, Brad Rees, and Joe Eaton. 2022. Rapids cugraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, 483–493.

[16] Alexandros V Gerbessiotis and Leslie G Valiant. 1994. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing* 22, 2 (1994), 251–267.

[17] The Khronos® SYCL™ Working Group. 29-03-2023. *SYCL 2020 Specification (revision 8) — registry.khronos.org.* https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

[18] Intel®. 2024. *oneAPI Base Toolkit.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html

[19] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 297–310. https://doi.org/10.14778/3157794.3157799

[20] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[21] David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann.

[22] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[23] U. Meyer and P. Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[25] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 622–636. https://doi.org/10.1145/3173162.3173180

[26] NVIDIA. 2024. *NVIDIA Nsight Compute.* https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[27] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[28] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[29] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting Label Propagation for Distributed Connected Components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18)*. Association for Computing Machinery, New York, NY, USA, 540–546. https://doi.org/10.1145/3159652.3159696

[30] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[31] TOP500.org. 2024. *November 2024 | TOP500.* https://top500.org/lists/top500/2024/11/

[32] Brandon Tran, Brennan Schaffner, Jason Sawin, Joseph M Myre, and David Chiu. 2020. Increasing the efficiency of GPU bitmap index query processing. In *International Conference on Database Systems for Advanced Applications*. Springer, 339–355.

[33] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[34] Kai Wang, Don Fussell, and Calvin Lin. 2021. A fast work-efficient SSSP algorithm for GPUs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/3437801.3441605

[35] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU. *ACM Trans. Archit. Code Optim.* 18, 2 (Feb. 2021). https://doi.org/10.1145/3444844

[36] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2851141.2851145

[37] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[38] Boyu Yang, Weiguo Zheng, Xiang Lian, Yuzheng Cai, and X Sean Wang. 2024. HERO: A Hierarchical Set Partitioning and Join Framework for Speeding up the Set Intersection Over Graphs. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–25.

[39] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Software* 48, 1 (Feb. 2022), 1:1–1:51. https://doi.org/10.1145/3466795

[40] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). https://doi.org/10.1145/3276491