HoloGraph: Bridging the Throughput Gap in Heterogeneous Graph Pattern Matching via Workload-Aware Steering

Haotian Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong, China
haotianma@link.cuhk.edu.cn

Wei-Chung Hsu
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong, China
hsuweichung@cuhk.edu.cn

Yeh-Ching Chung
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, Guangdong, China
ychung@cuhk.edu.cn

Abstract

Graph Pattern Matching (GPM) is a computationally demanding workload essential for modern data analytics. While emerging systems offer massive performance, they suffer from a fundamental throughput mismatch. CPU-centric systems leverage large host memory but are constrained by limited arithmetic instruction throughput during intensive set intersection operations. Conversely, GPU-accelerated systems offer massive compute power but are strictly bound by PCIe interconnect bandwidth. When processing large-scale graphs that exceed device memory, the data transfer throughput lags significantly behind the device's consumption rate, leaving compute engines starved.

This paper presents HoloGraph, a heterogeneous architecture designed to bridge this throughput gap. We introduce a workload-aware steering mechanism that routes lightweight, latency-sensitive tasks to the CPU and dense, compute-intensive intersections to the GPU, ensuring optimal hardware utilization. To resolve the interconnect bottleneck, we implement a metadata-driven protocol that leverages static GPU memory residency to transmit compact task metadata rather than raw subgraphs, effectively inverting the bandwidth limitation by compressing communication volume. Furthermore, we orchestrate a double-buffered, asynchronous pipeline to mask data-transfer latency by simultaneously generating host candidates and executing on the device. Evaluation shows that HoloGraph processes billion-scale graphs with sustaining high computational throughput to outperform state-of-the-art baselines by over an order of magnitude.

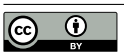
CCS Concepts

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Information systems → Graph-based database models**.

Keywords

Graph Pattern Matching, Heterogeneous System

ACM Reference Format:
Haotian Ma, Wei-Chung Hsu, and Yeh-Ching Chung. 2026. HoloGraph: Bridging the Throughput Gap in Heterogeneous Graph Pattern Matching via Workload-Aware Steering. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3797905.3807860



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3807860

1 Introduction

Graph Pattern Matching (GPM) is a critical component of contemporary data analytics, offering structural insights in bioinformatics [2, 14, 32], chemistry [12, 21], social networks [13, 18], and enhancing the expressive power of graph neural networks [11, 38, 40]. The computational complexity of matching increases exponentially with the pattern size ($|\mathcal{P}|$) and has been proven NP-Hardness [15, 16]. As "Big Data" repositories now contain billions of edges, this combinatorial explosion poses a severe challenge to modern computer architectures.

Recent GPM systems generally fall into two incompatible categories, forcing a trade-off between scalability and performance. The first category comprises CPU-centric systems [8, 10, 19, 20, 27, 33]. These architectures leverage massive host memory to accommodate large-scale graphs but are fundamentally restricted by their execution model. Designed for Task-Level Parallelism, generic CPU architectures lack the massive core counts required to process dense set intersections efficiently. While effective for complex control logic, this design forces the serialization of the fine-grained Data-Level Parallelism inherent in graph workloads. Consequently, the CPU remains compute-bound and unable to deliver the arithmetic throughput required for complex pattern matching.

The second category includes GPU-centric and heterogeneous systems [6, 7, 9, 15, 16]. While these systems offer massive SIMT parallelism, they are severely bandwidth-bound when processing graphs that exceed device memory capacity. To handle such workloads, the system must continuously migrate data from the host via the PCIe bus or coordinate intra-node transfers. This introduces a "throughput mismatch" where the GPU's massive compute engines are starving, stalled by the high latency and limited bandwidth of the interconnect rather than the device's High Bandwidth Memory (HBM). Consequently, the data transfer overhead nullifies the acceleration benefits, leaving a critical gap where no current architecture successfully harmonizes high-throughput computation with the massive data access requirements of large-scale graphs.

This performance gap stems from the fundamental Extend-Filter model [23, 36, 41], which decomposes GPM into two phases with diametrically opposed architectural affinities. The Extend phase involves irregular traversal, a control-intensive task that demands the sophisticated branch prediction of the host CPU to minimize

pipeline stalls. Conversely, the Filter phase performs dense set comparisons, a compute-intensive operation that dominates execution time and requires the massive throughput of SIMT architectures. This dichotomy renders homogeneous systems inefficient. A CPU-only approach suffers from compute scarcity during parallel intersections, while a GPU-only approach is constrained by control-flow divergence during traversal and the PCIe bandwidth bottleneck during data access.

This paper presents HoloGraph, a novel heterogeneous architecture designed to bridge the throughput gap in graph pattern matching. The system leverages the crucial insight that real-world graph workloads exhibit power-law characteristics to intelligently allocate hardware resources. Specifically, HoloGraph introduces three principal innovations.

- **Disaggregated Heterogeneous Coexecution:** We propose a heterogeneous architecture that partitions the graph pattern matching workload based on hardware affinity. We assign control-intensive traversal and latency-sensitive sparse intersections to the CPU, while specializing the GPU as a dedicated accelerator for dense, throughput-bound set operations. This separation maximizes host instruction throughput while ensuring the device is reserved for workloads that justify the offloading overhead.
- **Workload-aware Steering:** We develop a dynamic dispatch framework that exploits the power-law characteristics of real-world graphs. HoloGraph performs real-time cardinality analysis to retain lightweight tasks on the host and selectively routes only compute-intensive workloads to the GPU. This ensures that every processing unit operates solely on tasks that match its architectural strengths.
- **Metadata-Driven Protocol:** Addressing the fundamental PCIe bottleneck, we employ a task-aware caching strategy that pins high-degree neighbor lists in device memory. This enables a metadata-driven communication protocol where the host transmits compact task tuples rather than raw subgraphs, effectively inverting the bandwidth limitation by compressing the data transfer volume.

Our evaluation demonstrates that HoloGraph achieves the scalability of CPU-based systems while retaining the high-performance characteristics of in-memory GPU accelerators. It successfully processes graphs larger than the available device memory, preventing the crashes or execution timeouts observed in prior work and outperforming state-of-the-art baselines by over an order of magnitude.

2 Background

2.1 Preliminary

To understand the architectural challenges of Graph Pattern Matching (GPM), we first distill the workload into its fundamental operations. Formally, GPM is a subgraph isomorphism problem: given a large data graph $\mathcal{G} = (V, E)$ and a query pattern $\mathcal{P}$ (e.g., a triangle or a clique), the goal is to find all embeddings of $\mathcal{P}$ hidden within $\mathcal{G}$.

Quick start with the classic example *Triangle Counting*, where the system must find all sets of three vertices (u, v, w) that are mutually connected. Figure 1 illustrates one of the standard algorithmic

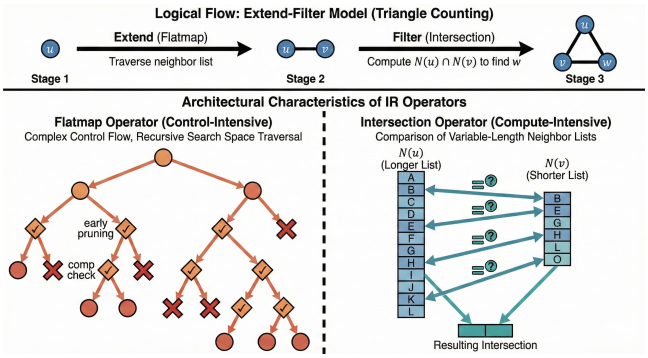


Figure 1: Extend-Filter Model for Graph Pattern Matching

approaches, known as the *Extend-Filter* model [23, 36, 41], which executes in steps:

(1) **Extend (Flatmap):** The system starts with a vertex (u) and attempts to extend it to an edge (u, v) by flatmap (the operator assigned to traverse its neighbor list).
(2) **Filter (Intersection):** To find a triangle, the system must figure out a third vertex w that connects to both u and v, by computing the intersection of the neighbor lists of u and v $(N(u) \cap N(v))$.

Typically, a query compiler translates the user input into a logical plan of these Intermediate Representation (IR) operators [23]. The backend processor then executes this plan by iterating over the graph's vertex or edge stream.

From an architectural standpoint, these operators exhibit orthogonal resource demands, creating a "resource affinity" dichotomy: The Flatmap phase is characterized by irregular memory access patterns and recursive backtracking. It is fundamentally control-intensive, relying heavily on the sophisticated branch prediction and out-of-order execution logic of modern CPUs to mitigate pipeline stalls during traversal. The Intersection phase, conversely, is compute-intensive. It involves the streaming comparison of sorted arrays, an operation that demands massive memory bandwidth and arithmetic throughput. While trivial for small sets, intersection becomes the dominant bottleneck for large-degree vertices, benefiting significantly from the massive parallelism of SIMT architectures.

Crucially, real-world graphs exhibit power-law distributions. This introduces extreme workload variance: a high-degree vertex may have millions of neighbors, necessitating high-throughput vector processing, while a low-degree vertex connects to only a few, favoring low-latency scalar execution. This skew exposes the inefficiency of homogeneous execution: a static assignment of tasks to either CPU or GPU inevitably results in under-utilization of resources for a significant portion of the dataset.

2.2 Existing GPM Systems

Contemporary GPM architectures employ distinct parallel execution models, optimizing either for logical control flow via Task-Level Parallelism (TLP) or for massive arithmetic throughput via Data-Level Parallelism (DLP).

Task Level Parallelism. Conventional CPU systems prioritize TLP to navigate the complex, irregular search space of subgraph

isomorphism. These frameworks map individual embedding search tasks to distinct CPU threads, leveraging the sophisticated branch prediction of modern processors to handle the irregular control flow inherent in the expansion phase. Peregrine [19, 20] exemplifies this approach by introducing a "pattern-aware" programming model. It analyzes the query structure to generate an efficient exploration plan, bypassing expensive canonicality checks by directly exploring subgraphs of interest. Addressing the computation redundancy caused by pattern symmetry, GraphZero [27, 28] utilizes group theory to generate symmetry-breaking constraints automatically. This ensures that each unique embedding is computed exactly once by a single thread without runtime overhead. Furthermore, GraphPi [33] advances this methodology by employing a performance prediction model to select the optimal combination of matching order and restriction sets. In this model, parallelism is achieved across independent search tasks, while the operations within each task (such as intersection) are typically executed sequentially or with limited vectorization within the thread context.

Data Level Parallelism. To exploit the massive arithmetic throughput of accelerators, GPU systems leverage DLP by mapping set operations to SIMT architectures. G^{2}Miner [7] adapts the traversal strategy from Breadth First Search (BFS) to Depth First Search (DFS) to better suit GPU memory hierarchies. It employs a code generator to create customized kernels that execute intersection operations in parallel across thousands of GPU threads. To address the memory capacity constraints that limit GPU adoption, PBE (Partition Based Enumeration) [15, 16] introduces a partition-centric approach. It divides large data graphs into chunks that fit within device memory and employs a "shared execution" strategy to enumerate subgraphs that span across partitions. These systems prioritize the parallel execution of the intersection operator, offloading the heavy arithmetic workload to the device while managing the control logic through specialized kernel scheduling.

2.3 Limitations

We contend that existing architectures fail to scale efficiently due to a fundamental throughput mismatch, where CPU solutions are strictly bound by arithmetic instruction throughput, while GPU and heterogeneous solutions are throttled by interconnect bandwidth.

CPU Compute Scarcity. The CPU is the traditional platform for GPM because its sophisticated branch predictors effectively handle the complex control flow of the Expansion phase. Modern CPU-based GPM systems, such as Peregrine [19, 20], GraphZero [27, 28], and GraphPi [33], leverage these capabilities to manage the complex control flow required for symmetry breaking and anti-edge constraints. However, the CPU suffers from inefficient parallelism during the intersection phase. When intersecting two arrays with millions of elements, the intersection becomes a massive data parallel task. A generic CPU core, even with vector extensions like AVX [17], lacks the massive thread count required to process these millions of elements in parallel. The hardware process GPM in task-level parallelism (one embedding per thread) effectively wastes the workload's intrinsic data parallelism. Figure 2 illustrates the performance gap measured by triangle counting, which nearly involves only the intersection. It exploits that the throughput of intersection processing is inefficient in multi-core CPU processing compared with GPU SIMT processing.

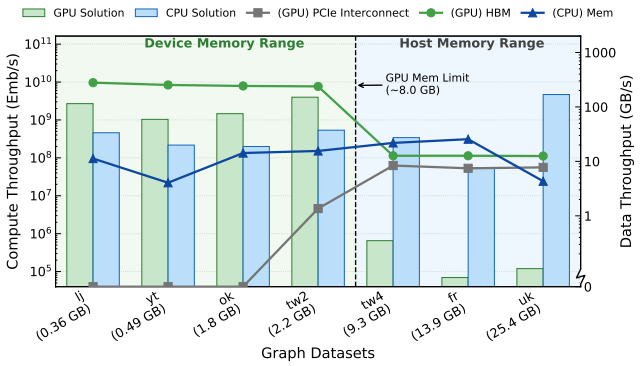


Figure 2: Processing throughput of existing architectures.

GPU Data Starvation. Conversely, GPUs possess the massive SIMT parallelism required to saturate intersection throughput but suffer from data starvation when graphs exceed device memory capacity. In these "out-of-core" scenarios, the system must continuously migrate data from the host via the PCIe bus [7, 15, 16]. As quantified in Figure 2, the system encounters a severe bandwidth cliff once the graph scale exceeds device memory capacity. When the graph scale increases to the host memory range, the effective data access rate plummets from the internal HBM bandwidth to the external PCIe limit. This bottleneck forces the GPU's massive SIMT engines into a state of data starvation, where execution stalls while awaiting operands from the host. Profiling confirms this throughput inversion: while the PCIe interconnect is fully saturated, GPU processing and HBM throughput drop significantly, demonstrating that system performance is strictly bottlenecked by communication latency rather than computational capacity.

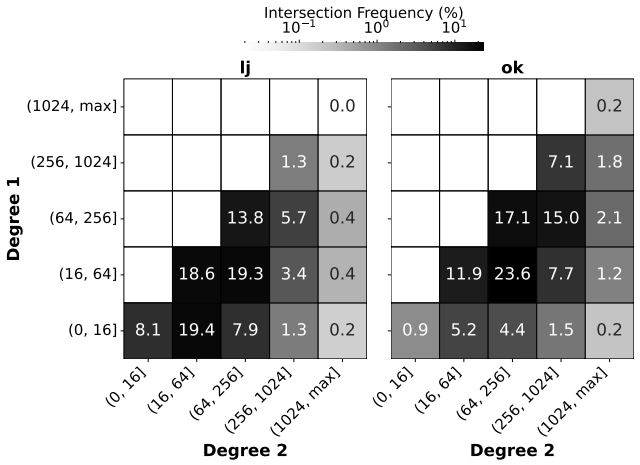


Figure 3: Intersection Frequency.

2.4 Motivation

This analysis demonstrates that existing architecture cannot efficiently accommodate the full spectrum of GPM workloads. While CPUs minimize latency for control-heavy Extend, they suffer from compute scarcity during dense set intersections. Conversely, GPUs provide sufficient computational throughput but are throttled by data starvation when processing large-scale graphs over the limited PCIe interconnect.

In addition, our approach resolves this resource under-utilization by exploiting the power-law distribution inherent in real-world graphs. Figure 3 visualizes the intersection workload distribution. We derive these frequencies by profiling canonical GPM patterns (Triangle, Rectangle, Pentagon) to capture representative multi-hop traversals. Symmetric operand pairs are aggregated to consolidate the cardinality space. Intersection workloads exhibit significant skewness: the vast majority involve small sets that do not benefit from massive parallelism (referred to as *sparse intersections*), while a minority of large sets account for the bulk of computational demand (referred to as *dense intersections*). Consequently, we introduce a workload-aware steering mechanism. To mitigate bus starvation, we pin high-degree neighbor lists in GPU memory, ensuring that dense intersections are executed on the GPU with the data residing there. Simultaneously, sparse intersections are routed to the CPU, leveraging its low-latency characteristics for fine-grained tasks.

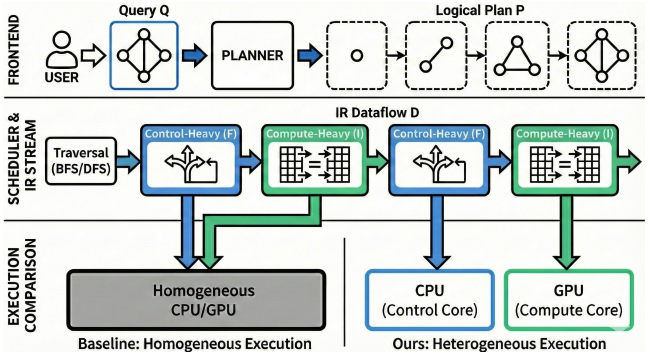


Figure 4: Homogeneous vs. Heterogeneous Execution

Figure 4 contrasts our heterogeneous execution model with traditional homogeneous approaches. We propose a disaggregated architecture that distributes workloads based on computational intensity. At the system level, the host and device operate in a cooperative pipeline. At the kernel level, each unit executes the form of parallelism at the task-level (CPU) and data-level (GPU) that best suits the hardware architecture. The goal of our design is to resolve the tension between compute scarcity and data starvation, thereby maximizing total system parallelism.

3 Heterogeneous Coexecution

3.1 Overview

HoloGraph is architected as a throughput-oriented heterogeneous co-execution system. It establishes the Host CPU and Device GPU as distinct semantic engines, integrating Workload-Aware Steering with Asynchronous Pipelining to maximize system-wide utilization.

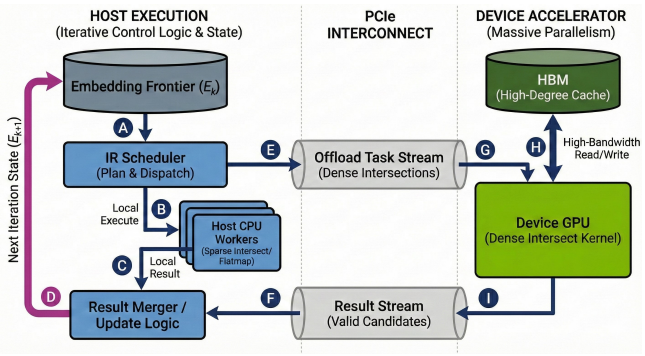


Figure 5: High-level Design of HoloGraph

As depicted in Figure 5, the architecture is tripartite, comprising three primary components: the *Host Controller*, the *Device Accelerator*, and the *Asynchronous Communication Interface*.

The host controller functions as the system's primary orchestrator, hosting the full data graph $\mathcal{G}$ in system DRAM, iteratively processing embeddings from the frontier, scheduling operands for embedding expansion based on the high-level logical plan, and dispatching the task to compute units. Specifically, the host controller identifies and dispatches low-workload, inefficiently parallel tasks (flatmap and sparse intersection) to CPU cores, and steers compute-intensive tasks (dense intersection) to the device worker via the PCIe bus.

The device accelerator serves exclusively as a high-throughput engine for dense set intersections. To avoid the PCIe bottleneck, we leverage GPU High Bandwidth Memory (HBM) to statically cache high-degree neighbor lists demanded by dense intersections. This resident data enables the device to execute intersections without waiting for data transfer, eliminating interconnect latency.

The asynchronous communication interface employs a bidirectional buffer over the PCIe bus to overlap latency and prevent the communication throughput bottleneck. By orchestrating a continuous pipeline, we ensure that the host and the device independently execute batches of tasks simultaneously. The communication latency of task dispatching and result returning is overlapped by execution.

3.2 The Control-Compute Workflow

The primary responsibility of the host controller is to execute pattern-aware exploration logic iteratively on embeddings from the frontier, initiated with an embedding frontier E_0 derived from vertex enumeration.

For each embedding in the active frontier, the host evaluates the subsequent Intermediate Representation (IR) operator to determine the execution path. Latency-sensitive workloads such as flatmap and sparse intersection are processed locally for immediate scalar execution (Steps A-C). Conversely, dense intersections are offloaded to the GPU via a producer-consumer queue. Crucially, this dispatch mechanism is non-blocking: the host preserves the context of offloaded tasks and immediately processes the remaining tasks in the frontier (Step A). Upon retrieving completion signals from the device, the host resumes these contexts and promotes valid

embeddings to the next processing stage (Steps D–F). We rigorously define the cardinality threshold distinguishing sparse from dense workloads in §4.

The device accelerator (GPU) is specialized as a high-throughput engine for dense set intersections ($Set_a \cap Set_b$) (Step G-I). It continuously polls the producer-consumer queue to retrieve independent tasks, and remains unaware of the dispatch latency until the queue is empty. To maximize interconnect efficiency, the device receives only lightweight task metadata, using these identifiers to index the full neighbor sets resident in High Bandwidth Memory (HBM). After the computation, the results are asynchronously written back to a pinned buffer in host memory.

4 Workload-aware Steering

4.1 Workload Characterization

To optimize resource allocation, we first characterize the computational intensity of set intersection based on operand cardinality. We observe that the throughput profile diverges significantly based on input size, as illustrated in Figure 6.

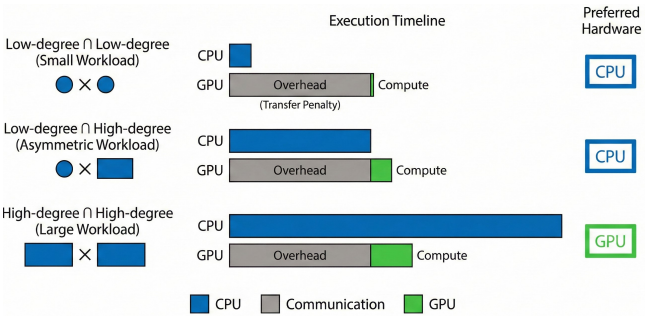


Figure 6: Compute Characteristic of Intersections

First, intersections between two low-degree sets fail to saturate the GPU's massive thread capacity. In this latency-bound scenario, the overhead of PCIe interconnect significantly outweighs the computational cost. GPU offloading is inefficient at the small-workload intersection. Second, asymmetric intersections (one low-degree set, one high-degree set) remain inefficient on the GPU. While GPU parallelism accelerates the comparison of intersections, the cost of offloading the large operand across the bus negates the benefit. These workloads are highly amenable to CPU execution using galloping binary search, which exploits the size disparity to reduce complexity from linear to logarithmic time ($O(|A| \log |B|)$), which generally performs better than the GPU solution with interconnect overhead. Finally, intersections between two high-degree sets provide sufficient workload to fully occupy the GPU's SIMT lanes. This allows the GPU to effectively reduce computation time through massive thread-level parallelism.

Consequently, we formally define the Steering Policy to determine the optimal execution unit for each intersection task. We introduce a cardinality threshold τ to distinguish between high-degree set ($|Set| > \tau$) and low-degree ($|Set| \leq \tau$). Accordingly, we classify an intersection task $Intersect(Set_a, Set_b)$ as a dense intersection if and only if both sets exceed this threshold:

$$\text{Intersection}(Set_a, Set_b) = \begin{cases} \text{Sparse} & \max(|Set_a|, |Set_b|) < \tau \\ \text{Dense} & \text{otherwise} \end{cases}$$

Under this definition, only dense intersections are offloaded to the GPU to maximize throughput, while all sparse intersections are retained on the host to minimize latency.

4.2 Steering Threshold

Existing architectures remain fundamentally constrained by PCIe bandwidth limits, incurring a substantial latency penalty from frequent host-device communication. To balance the computational benefits of SIMT execution against the interconnect overhead, we employ a greedy, capacity-aware steering mechanism, which derives the steering threshold τ by reconciling graph scale with the GPU's physical memory constraints.

Algorithm 1 Greedy Capacity-Aware Steering

Require: Graph $\mathcal{G}(V, E)$, Device Memory Capacity $\mathcal{M}$
Ensure: Steering Threshold τ

Initialize frequency table $H \leftarrow \{0, \ldots, 0\}$
$d_{max} \leftarrow 0$
// Phase 1: Build Degree Frequency Table
for $v \in V$ **do**
  $d \leftarrow \deg(v)$
  $H[d] \leftarrow H[d] + 1$
  $d_{max} \leftarrow \max(d_{max}, d)$
end for
$\tau \leftarrow d_{max}$
// Phase 2: Greedy Capacity Allocation
while $\tau \geq 0$ **do**
  $usage \leftarrow H[\tau] \times \tau$
  if $\mathcal{M} \geq usage$ **then**
    $\mathcal{M} \leftarrow \mathcal{M} - usage$
    $\tau \leftarrow \tau - 1$
  else
    break {Capacity saturated}
  end if
end while
return τ

As formalized in Algorithm 1, we employ a greedy capacity allocation strategy that prioritizes the residency of dense neighbor lists. The system iterates through vertices in descending order of degree, pinning their adjacency arrays to device HBM until the memory budget is exhausted. We subsequently define the steering threshold τ as the degree of the last vertex successfully cached. This construction provides a deterministic runtime guarantee: since the dispatcher restricts GPU offloading exclusively to dense intersections, both operand sets are provably resident in device memory (as validated in §5). Consequently, HoloGraph eliminates the overhead of raw data migration, limiting PCIe traffic to the transmission of lightweight task metadata.

Furthermore, this mechanism is inherently adaptive in scenarios where the graph scale allows full residency (i.e., the entire dataset fits within device memory) and the threshold naturally converges to

zero ($\tau \to 0$). HoloGraph automatically caches the complete graph to maximize throughput without requiring manual reconfiguration.

5 Metadata-Driven Protocol

By leveraging the static residency of high-degree neighbor lists in device memory, we implement a metadata-driven communication protocol. We observe a critical invariant enforced by our steering logic: any operand set required for a GPU-offloaded intersection is, by definition of the threshold ($\min(|Set|) > \tau$), already resident in HBM. Consequently, we eliminate the redundant transmission of raw vertex arrays across the PCIe bus. Instead, the host dispatches task metadata as illustrated in Figure 7. This mechanism transforms the communication model from a data-bound transfer of megabytes into a stream of bytes, ensuring that interconnect bandwidth no longer limits the GPU's computational throughput.

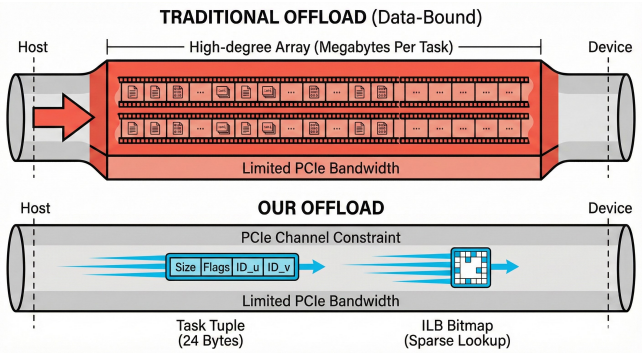


Figure 7: Traffic Alleviation by Metadata-Driven Protocol

5.1 Ephemeral Sets Management

A fundamental challenge in traffic reduction lies in managing the ephemeral intermediate data generated during multi-step query execution. For initial intersections ($N(u) \cap N(v)$), transferring simple `Vertex IDs` is sufficient, as the device can directly index the resident neighbor lists from its static cache. However, complex GPM queries frequently necessitate chained intersections (e.g., $(N(u) \cap N(v)) \cap N(w)$). Our heterogeneous pipeline is architected for atomic dispatch, executing strictly one intersection per task to maintain statelessness and load balance. Consequently, the intermediate result $S' = N(u) \cap N(v)$ is an ephemeral set: it is anonymous, lacking a static `Vertex ID`, and exists only transiently within the execution context. Naively transferring S' back to the host solely to re-upload it for the subsequent intersection ($S' \cap N(w)$) would effectively reintroduce the very bandwidth bottleneck our architecture seeks to eliminate.

To address the referencing of these anonymous ephemeral sets, we introduce the Index Lookup Bitmap (ILB). We redefine the task metadata tuple as a compact descriptor: `Metadata Size, Flags, Vertex ID, ILB`. The ILB serves as a bitmask that enables the device to reconstruct specific ephemeral subsets without raw data transfer. This mechanism exploits a critical data dependency invariant that any intermediate set S' generated during a GPM query is a subset of its input operands. Since our steering mechanism in §4 restricts GPU execution exclusively to dense Intersections, the parent superset is guaranteed to be statically cached in HBM. Therefore, we can uniquely identify any ephemeral set S' using {`Vertex ID`, `ILB`}, where the `Vertex ID` points to the base superset in device memory, and the `ILB` masks the invalid elements.

5.2 Metadata Management

We utilize the Index Lookup Bitmap (ILB) in two distinct phases of the execution lifecycle to minimize data movement and redundant computation.

Candidate Filtering: Threads within a warp execute the intersection logic on the resident neighbor lists as established in §6.2. Upon detecting a potential match, the thread validates the candidate against the active input ILB. If the corresponding bit is unset, the candidate is deemed invalid and immediately discarded. To mitigate control divergence, we employ an `is_subset` flag. This signal allows the kernel to bypass the ILB check entirely when the input operand is a complete neighbor list, thereby eliding unnecessary memory lookups and minimizing branch overhead.

Output Generation: Simultaneously, the kernel constructs the ILB for the result set if the output cardinality exceeds τ. ILB encodes valid matches relative to their indices in the shorter source array (the Reference Parent). By transmitting this compact bitset back rather than integer arrays, we effectively compress the intermediate results. We further optimize this with a `will_intersect` flag: if the logical plan indicates the result will not act as an operand for subsequent intersections, we suppress ILB generation entirely. Collectively, this Metadata-Driven Protocol reduces host-device data transfer volume by orders of magnitude.

6 GPM Parallelism

6.1 Pipelining Parallelism

In addition to the interconnect bandwidth, communication latency also presents a critical bottleneck in decoupled architectures. System throughput is severely throttled if the host stalls awaiting device completion or if the device idles pending task dispatch. HoloGraph mitigates this overhead by implementing an asynchronous pipeline, detailed in Figure 8.

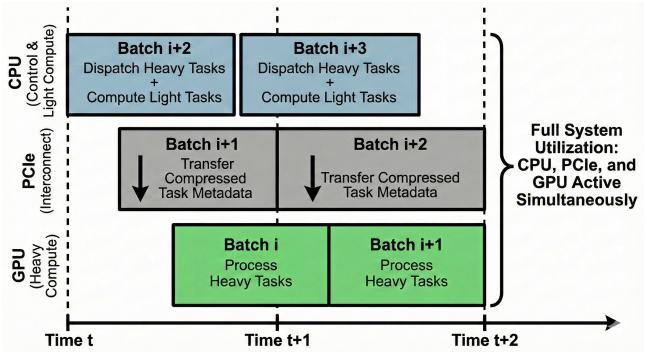


Figure 8: Pipeline Parallelism of HoloGraph

The execution pipeline establishes a continuous stream of active batches, orchestrated through three overlapping stages to maximize

resource utilization. As depicted in Figure 8, at any given instant t, the system operates as follows: *1) Host Control (Batch N + 2).* The CPU first identifies the IR executor of batch $N + 2$ from the frontier embedding. Then it executes flatmap operations and sparse intersections locally and dispatches tasks to the GPU for dense intersections. *2) PCIe Interconnect (Batch N + 1).* The DMA engine streams the task metadata for the previously packed batch to the device. This operation effectively masks data movement latency behind the host's computation [15, 16, 25]. *3) GPU Compute (Batch N).* The GPU executes the intersection kernel for the resident batch using the task metadata.

This three-stage design ensures that the CPU, PCIe interconnect, and GPU operate concurrently, effectively eliminating the synchronization bubbles characteristic of synchronous offloading schemes.

6.2 Intersection Parallelism

We exploit intersection parallelism through three algorithmic primitives, each tailored to the specific cardinality of the workload and the architectural strengths of the underlying hardware.

To maximize host throughput, we leverage AVX512 vector extensions to implement a distinct kernel for the small workload (low-degree ∩ low-degree) and asymmetric workload (low-degree ∩ high-degree) intersection. For small workloads, we execute a SIMD-optimized Merge Path algorithm [17]. By packing multiple 32-bit neighbor IDs into 512-bit vector registers, this primitive performs parallel comparisons in a single cycle. This effectively saturates the CPU's superscalar instruction pipeline, transforming the control-heavy linear scan into a throughput-oriented vector operation. For asymmetric workloads, linear scanning is computationally wasteful. We employ SIMD-Accelerated Galloping Binary Search [24]. This technique exploits the size disparity to logarithmically prune the search space of the massive array ($O(|A| \log |B|)$). By skipping strictly dominated ranges, it minimizes unnecessary cache line fills and optimizes latency for these asymmetric workloads.

For dense intersections offloaded to the device, we adopt a Warp Parallel Binary Search strategy [7] to maximize SIMT efficiency. The kernel maps each independent intersection task to a single warp, ensuring high hardware occupancy. Within this context, threads cooperatively perform coalesced loads to fetch blocks from the smaller neighbor list (the search keys) and concurrently execute binary searches against the longer list (the search space) resident in HBM.

7 Evaluation

We evaluate HoloGraph to answer four research questions:
RQ1: Does HoloGraph outperform state-of-the-art systems in execution time?
RQ2: How effectively does the memory hierarchy sustain data throughput in our system?
RQ3: How does the workload-aware steering strategy contribute to the performance?
RQ4: How well does the metadata-driven protocol eliminate redundant transfers to prevent bandwidth exhaustion?

Graph	# Nodes	# Edges	Size
Youtube (yt)	7M	114.2M	0.49GB
LiveJournal (lj)	4.0M	34.7M	0.36GB
Orkut (ok)	3.1M	117.2M	1.8GB
Twitter20 (tw2)	21.3M	265M	2.2GB
Twitter40 (tw4)	41.7M	1202.5M	9.3GB
Friendster (fr)	65.6M	1806M	13.9GB
UK-2007 (uk)	105.9M	3301.2M	25.4GB

Table 1: Graph Datasets

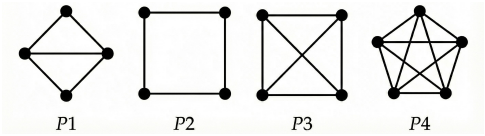


Figure 9: Patterns used in the evaluation

7.1 Experiment Setup

We evaluate HoloGraph on a heterogeneous server equipped with dual-socket Intel Xeon Silver 4210R processors (40 threads, 96 GB memory) and an NVIDIA RTX 4000 GPU (8 GB device memory). We benchmark against GraphZero and G^{2}Miner to represent state-of-the-art CPU and GPU architectures, respectively. To isolate the efficiency of the runtime engine, we enforce identical logical query plans across all systems. This methodology ensures that observed performance differences are strictly attributable to hardware utilization rather than to variations in query optimization or symmetry-breaking strategies [15, 16, 33]. All systems execute identical logical plans for the four structural patterns shown in Figure 9, ensuring that performance differentials are strictly attributable to runtime execution throughput rather than query planning variations. We utilize a diverse suite of real-world datasets [3–5, 22, 39], as detailed in Table 1.

7.2 End-to-End Performance Analysis

Figure 10 presents the end-to-end execution time of HoloGraph compared to GraphZero and G^{2}Miner across varying dataset sizes and pattern complexities. We denote instances exceeding a 36-hour execution limit with "T" (Bars with all baselines crashed by timeout were omitted). The results demonstrate that HoloGraph provides the most robust performance profile, effectively bridging the gap between high-speed in-memory execution and scalable out-of-core processing.

On datasets that fit entirely within the GPU's memory, G^{2}Miner generally exhibits the lowest latency. This is expected, as G^{2}Miner exploits the high bandwidth of HBM (High Bandwidth Memory) without incurring PCIe data-transfer overhead. HoloGraph remains competitive in this scale, with only marginal slowdowns attributed to runtime scheduling and metadata management overheads. This confirms that our hybrid architecture incurs minimal penalty even when fully in-memory execution is possible.

The benefits of HoloGraph become decisive on billion-scale graphs ($tw4$, fr, uk), which exceed GPU capacity. G^{2}Miner fails to execute any patterns on these datasets due to memory exhaustion.

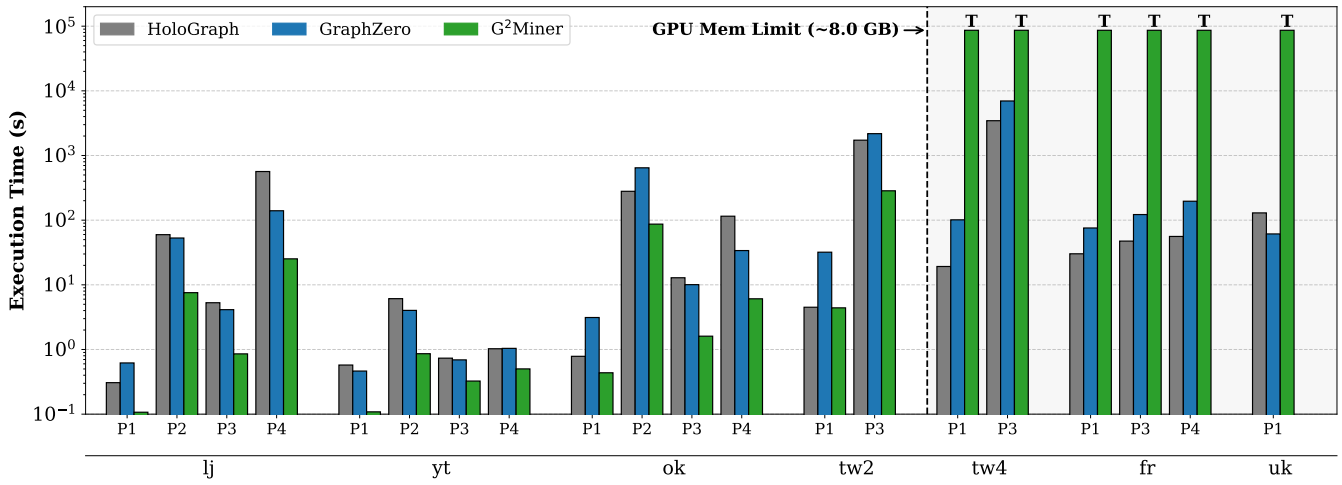


Figure 10: Overall performance of HoloGraph, GraphZero, and G²Miner.

In contrast, HoloGraph successfully scales to all inputs, outperforming the CPU-based GraphZero by up to an order of magnitude (e.g., *tw*4 on P_3). This speedup confirms the efficacy of our pipeline in hiding PCIe latency by overlapping data transfer with compute-intensive intersection kernels.

The *uk* dataset represents a pathological case due to its sparsity. In P_1, HoloGraph exhibits performance parity with GraphZero rather than a significant speedup. The low arithmetic intensity of sparse intersections on *uk* is insufficient to saturate the GPU's compute units or to effectively hide PCIe transfer latency. This limitation highlights the necessity of our adaptive steering, which prevents performance degradation by falling back to CPU execution when offloading costs outweigh acceleration benefits.

Hardware Utilization. Table 2 details the active CPU and GPU utilization rates across multiple large-scale graphs. These internal profiling results confirm high overall system saturation. By effectively overlapping host and device computations, the asynchronous pipeline maintains high resource utilization across varying graph topologies and pattern complexities.

Graph	Hardware	P1	P2	P3	P4
Twitter20	GPU	82.21%	100.00%	99.62%	99.96%
	CPU	49.31%	98.77%	99.77%	98.58%
Twitter40	GPU	94.45%	99.83%	78.40%	83.78%
	CPU	73.04%	99.45%	95.79%	95.59%
Friendster	GPU	89.53%	95.21%	92.38%	98.23%
	CPU	58.01%	75.64%	76.39%	92.34%
UK-2007	GPU	86.48%	74.02%	69.48%	71.42%
	CPU	70.34%	98.37%	97.73%	95.78%

Table 2: Hardware utilization of HoloGraph

7.3 Data Throughput Analysis

To understand the architectural sources of HoloGraph's performance, we analyze the effective data throughput across the memory hierarchy (Host Memory, PCIe, and GPU HBM) in Figure 11. We present results for two representative patterns: P_1 and P_2. P_1 is intersection-intensive and compute-bound and P_2 balances flat-map expansions with intersection filtering.

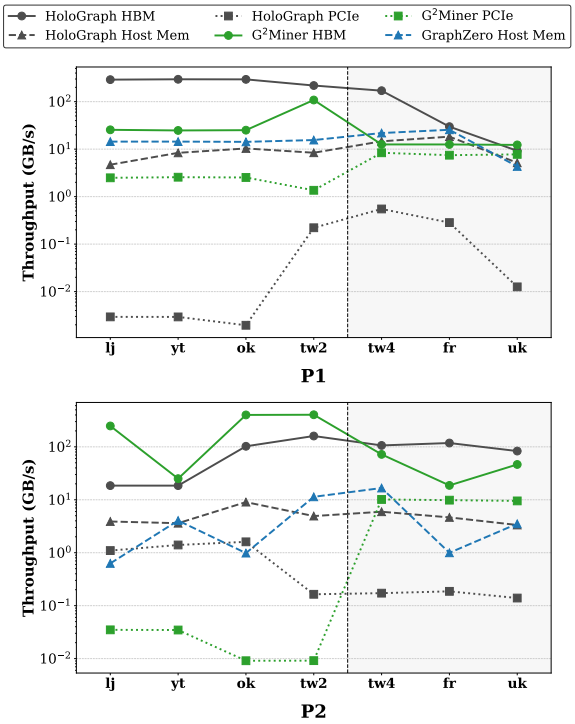


Figure 11: Overall data throughput of Systems.

A critical metric for GPU accelerators is the utilization of High Bandwidth Memory (HBM). HoloGraph consistently sustains high HBM throughput, significantly outperforming the baselines on large-scale datasets. On the $tw4$ and fr graphs, our system achieves an effective HBM throughput that is orders of magnitude higher than G^{2}Miner [7]. This indicates that HoloGraph successfully maintains a high occupancy of active warps on the GPU. Unlike G^{2}Miner, which suffers from memory thrashing or capacity overflow on large graphs, our pipeline ensures that the GPU always processes valid, resident subgraphs.

HoloGraph maintains consistently low PCIe throughput (0.1-1.0 GB/s), utilizing less than 10% of the available PCIe bandwidth. This demonstrates that HoloGraph effectively eliminates the PCIe bottleneck, decoupling execution performance from interconnect bandwidth. Unlike G^{2}Miner, which saturates the bus by streaming raw subgraphs, HoloGraph's metadata-driven protocol filters redundancy at the source.

HoloGraph maintains host memory throughput parity with the CPU-only baseline [27], implying that host threads remain active rather than stalling on accelerator feedback. This demonstrates effective latency masking, where CPU-side candidate generation is continuously overlapped with device-side intersection to prevent resource idling.

7.4 Efficacy of Workload-Aware Steering

To quantify the benefits of our workload-aware scheduling (WAS), we conduct an ablation study comparing the full system (-WAS) against a version that indiscriminately offloads all tasks (-Only). Figure 12 presents the execution time and throughput breakdown for patterns P_1 and P_3 on the $tw4$ and fr datasets.

As shown in Figure 12(a), the WAS strategy is critical for performance stability. The ablation baseline (-Only) fails to complete execution within the time limit for $tw4$ on both P_1 and P_3. This failure stems from inefficient sparse offloading that sends low-degree intersection tasks to the GPU, which incurs synchronization and transfer overheads that outweigh the computational speedup. By filtering these sparse tasks and processing them on the host CPU, the system achieves orders-of-magnitude speedups and ensures robust scalability.

Figure 12(b) reveals the architectural root of this performance divergence. WAS sustains HBM throughputs exceeding 100 GB/s. This high efficiency confirms that the steering policy successfully selects dense, computationally intensive workloads that generate consecutive memory accesses. These access patterns enable effective memory coalescing on the GPU, maximizing the utility of the high-bandwidth memory. In contrast, the baseline exhibits negligible HBM throughput ($< 10^{-3}$ GB/s), indicating that the GPU is stalled processing fragmented, fine-grained memory requests that fail to saturate the memory controller.

A counter-intuitive result appears in the PCIe throughput, where the baseline (-Only) achieves higher PCIe throughput. However, this represents inefficient saturation as we analyze in §7.3: The interconnect is bottlenecked with metadata and small task headers for sparse workloads that yield minimal compute. Conversely, workload-aware steering maintains a low PCIe profile while delivering superior end-to-end performance. This confirms that WAS

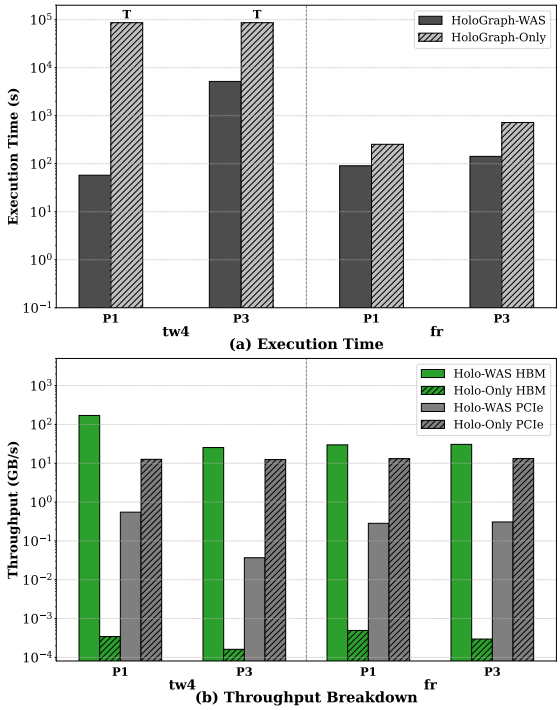


Figure 12: Impact of Workload-Aware Steering (WAS).

acts as an intelligent filter, offloading only tasks that require significant computation. This prevents interconnect thrashing and ensures PCIe bandwidth is allocated to workloads that benefit from acceleration.

7.5 Efficacy of Metadata-Driven Protocol

To isolate the impact of our communication optimization, we evaluate the system with and without the Metadata-Driven Protocol (MDP). Figure 13 illustrates the execution status and PCIe throughput on the largest datasets.

The results on the fr dataset demonstrate the necessity of MDP for bandwidth conservation. When the MDP is disabled, the PCIe becomes fully saturated, reaching the interconnect limit. This indicates the interconnect is saturated by redundant data transfers, causing pipeline congestion and execution timeouts. The significantly reduced bandwidth profile confirms that MDP effectively filters redundancy at the source, restricting interconnect traffic to strictly compressed metadata.

The behavior on $tw4$ reveals a critical scalability issue. The ablation case exhibits anomalously low PCIe throughput, which is a symptom of execution instability rather than efficiency. Without MDP's filtering, the aggressive data streaming overwhelms device memory, triggering Out-Of-Memory (OOM) exceptions and preventing the system from reaching steady-state utilization. This shows that MDP balances data supply with memory capacity to ensure scalable execution.

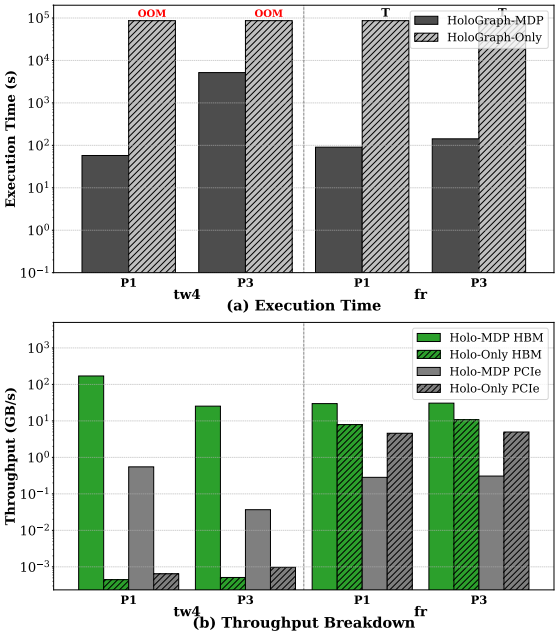


Figure 13: Impact of Metadata-Drive Protocol (MDP).

8 Related Work

Graph pattern-matching systems have been developed for several years; prior studies have explored many different acceleration approaches, including algorithmic pruning, optimal search plans, and faster execution with a more suitable parallel architecture. Some of the techniques are orthogonal to our study, and they can be combined to achieve better performance.

Algorithmic Pruning and Compile-Time Optimization. To mitigate the combinatorial explosion inherent in subgraph matching, modern frameworks increasingly leverage static analysis and theoretical reductions to contract the search space prior to execution. While early systems utilized intermediate representations to decouple algorithmic definition from backend execution [26], recent work has integrated group theory directly into optimizing query compilers. These approaches automate symmetry breaking, generating constraints that eliminate automorphism-induced redundant computations [27, 33]. Concurrently, the adoption of Worst-Case Optimal Join (WCOJ) algorithms has shifted the focus from traditional binary joins to multi-way intersection strategies [29, 30]. By exploiting generalized hypertree decompositions and SIMD parallelism, these techniques demonstrate significant speedups over conventional graph engines, extending efficacy to complex patterns involving negative or optional edges [1, 31].

Architecture-Aware Parallel Traversal. Optimizing graph pattern matching on parallel hardware requires balancing massive concurrency with limited memory resources and divergent control flow. Although initial GPU-accelerated methods relied on bulk-synchronous BFS traversals [9], the high memory footprint of intermediate frontiers necessitated a shift toward depth-first strategies. Consequently, recent architectures employ stack-based backtracking and hybrid BFS-DFS schedules to minimize memory consumption while maximizing thread occupancy [7, 34, 37]. To address the load imbalance caused by power-law graph distributions, systems now routinely incorporate dynamic work-stealing mechanisms and relaxed execution ordering [35, 37]. Beyond general-purpose GPUs, hardware-software co-designs have emerged to further reduce latency, utilizing specialized processing elements to accelerate isomorphism checks and candidate filtering [10].

9 Conclusion and Future Work

This paper presents HoloGraph, a heterogeneous architecture that bridges the fundamental throughput gap in Graph Pattern Matching by resolving the tension between CPU compute scarcity and GPU data starvation. By synergizing Workload-Aware Steering with a Metadata-Driven Protocol, we decouple execution based on hardware affinity to eliminate redundant data transfer with routing sparse tasks to the CPU and dense intersections to the GPU. Evaluation results demonstrate that HoloGraph sustains HBM throughputs exceeding 100 GB/s while saturating the PCIe bus by less than 10%. Consequently, it delivers performance gains of over an order of magnitude against state-of-the-art baselines on large-scale graphs.

In future work, we aim to address the diminishing returns observed in highly sparse intersections (e.g., the uk dataset), where the fixed overhead of heterogeneous coordination can outweigh acceleration benefits. We plan to investigate more fine-grained pipeline optimizations and develop a dynamic steering model that adapts the offloading threshold to the embedding frontier's instantaneous sparsity, ensuring robust performance across all graph distributions. Additionally, HoloGraph will support graph pattern matching on dynamic graphs, enabling broader practical use.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive feedback and rigorous evaluation, which significantly improved the quality of this paper. We also acknowledge the use of generative AI tools to assist in refining the text and conceptual illustrations. This work was supported by Huawei Technologies Co., Ltd., China, under contract No. TC20220913039, and by the Pengcheng Peacock Plan under Grants 2025TC0017 and 2023TA0033.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. doi:10.1145/3129246

[2] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S. Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (07 2008), i241–i249. doi:10.1093/bioinformatics/btn163

[3] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web* 12, 2, Article 12 (June 2018), 26 pages. doi:10.1145/3160017

[4] 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* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. doi:10.1145/1963405.1963488

[5] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. doi:10.1145/988672.988752

[6] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 32, 12 pages. doi:10.1145/3190508.3190545

[7] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 857–877. https://www.usenix.org/conference/osdi22/presentation/chen

[8] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 378–391. doi:10.1145/3447818.3460359

[9] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: an efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.* 13, 8 (April 2020), 1190–1205. doi:10.14778/3389133.3389137

[10] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: a pattern-aware accelerator for graph pattern mining. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 581–594. doi:10.1109/ISCA52012.2021.00052

[11] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can graph neural networks count substructures?. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 871, 13 pages.

[12] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1036–1050. doi:10.1109/TKDE.2005.127

[13] G.W. Flake, S. Lawrence, C.L. Giles, and F.M. Coetzee. 2002. Self-organization and identification of Web communities. *Computer* 35, 3 (2002), 66–70. doi:10.1109/2.989932

[14] Joshua A. Grochow and Manolis Kellis. 2007. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology* (Oakland, CA, USA) *(RECOMB'07)*. Springer-Verlag, Berlin, Heidelberg, 92–106.

[15] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1067–1082. doi:10.1145/3318464.3389699

[16] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2022. Exploiting Reuse for GPU Subgraph Enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2022), 4231–4244. doi:10.1109/TKDE.2020.3035564

[17] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1587–1602. doi:10.1145/3183713.3196924

[18] Yanqing Hu, Shenggong Ji, Yuliang Jin, Ling Feng, H. Eugene Stanley, and Shlomo Havlin. 2018. Local structure can identify and quantify influential global spreaders in large scale social networks. *Proceedings of the National Academy of Sciences* 115, 29 (2018), 7468–7472. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1710547115 doi:10.1073/pnas.1710547115

[19] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 16 pages. doi:10.1145/3342195.3387548

[20] Kasra Jamshidi and Keval Vora. 2021. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 1–10. doi:10.1145/3469379.3469381

[21] Hisashi Kashima, Hiroto Saigo, Masahiro Hattori, and Koji Tsuda. 2010. *Graph kernels for chemoinformatics*. IGI Global, 1–15. doi:10.4018/978-1-61520-911-8.ch001

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. doi:10.1145/1772690.1772751

[23] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 53–69. https://www.usenix.org/conference/atc23/presentation/lai

[24] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Softw. Pract. Exper.* 46, 6 (June 2016), 723–749. doi:10.1002/spe.2326

[25] Hang Liu and H. Howie Huang. 2017. Graphene: fine-grained IO management for graph computing. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 285–299.

[26] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. 2024. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques* (Atlanta, GA, USA) *(PACT '21)*. IEEE Press, 289–303. doi:10.1109/PACT52795.2021.00028

[27] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Oper. Syst. Rev.* 55, 1 (June 2021), 21–37. doi:10.1145/3469379.3469383

[28] Daniel Mawhirter and Bo Wu. 2019. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 509–523. doi:10.1145/3341301.3359633

[29] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. doi:10.14778/3342263.3342643

[30] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. doi:10.1145/3180143

[31] Sungwoo Park, Seyeon Oh, and Min-Soo Kim. 2025. cuMatch: A GPU-based Memory-Efficient Worst-case Optimal Join Processing Method for Subgraph Queries with Complex Patterns. *Proc. ACM Manag. Data* 3, 3, Article 143 (June 2025), 28 pages. doi:10.1145/3725398

[32] N. Pržulj, D. G. Corneil, and I. Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (02 2006), 974–980. arXiv:https://academic.oup.com/bioinformatics/article-pdf/22/8/974/48841561/bioinformatics_22_8_974.pdf doi:10.1093/bioinformatics/btl030

[33] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 100, 14 pages.

[34] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2, Article 181 (June 2023), 26 pages. doi:10.1145/3589326

[35] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 209–224. https://www.usenix.org/conference/atc21/presentation/trigonakis

[36] Haixun Wang, Bolin Ding, Jeffrey Xu Yu, Philip S. Yu, and Jiefeng Cheng. 2008. Fast Graph Pattern Matching . In *2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, Los Alamitos, CA, USA, 913–922. doi:10.1109/ICDE.2008.4497500

[37] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 53, 13 pages.

[38] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *CoRR* abs/1810.00826 (2018). arXiv:1810.00826 http://arxiv.org/abs/1810.00826

[39] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. doi:10.1145/2350190.2350193

[40] Jiaxuan You, Jonathan Gomes Selman, Rex Ying, and Jure Leskovec. 2021. Identity-aware Graph Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence* 35 (05 2021), 10737–10745. doi:10.1609/aaai.v35i12.17283

[41] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1, Article 60 (March 2024), 29 pages. doi:10.1145/3639315