RESEARCH-ARTICLE

gHyPart: GPU-friendly End-to-End Hypergraph Partitioner

ZHENLIN WU, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

HAOSONG ZHAO, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

HONGYUAN LIU, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

WUJIE WEN, NC State University, Raleigh, NC, United States

JIAJIA LI, NC State University, Raleigh, NC, United States

Published: 21 March 2025
Online AM: 10 January 2025
Accepted: 25 December 2024
Revised: 04 November 2024
Received: 01 July 2024

gHyPart: GPU-friendly End-to-End Hypergraph Partitioner

ZHENLIN WU, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
HAOSONG ZHAO, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
HONGYUAN LIU, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
WUJIE WEN, North Carolina State University at Raleigh, Raleigh, United States
JIAJIA LI, North Carolina State University at Raleigh, Raleigh, United States

Hypergraph partitioning finds practical applications in various fields, such as high-performance computing and circuit partitioning in VLSI physical design, where high-performance solutions often demand substantial parallelism beyond what existing CPU-based solutions can offer. While GPUs are promising in this regard, their potential in hypergraph partitioning remains unexplored. In this work, we first develop an end-to-end deterministic hypergraph partitioner on GPUs, ported from state-of-the-art multi-threaded CPU work, and identify three major performance challenges by characterizing its performance. We propose the first end-to-end solution, gHyPart, to unleash the potentials of hypergraph partitioning on GPUs. To overcome the challenges of GPU thread underutilization due to imbalanced workload, long critical path, and high work complexity due to excessive operations, we redesign GPU algorithms with diverse parallelization strategies thus expanding optimization space; to address the challenge of no one-size-fits-all implementation for various input hypergraphs, we propose a decision tree-based strategy to choose a suitable parallelization strategy for each kernel. Evaluation on 500 hypergraphs shows up to 125.7× (17.5× on average), 640.0× (24.2× on average), and 171.6× (1.4× on average) speedups over two CPU partitioners and our GPU baseline gHyPart-B, respectively.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Massively parallel algorithms;**

Additional Key Words and Phrases: Hypergraph partitioning, GPU, parallelization strategies

ACM Reference Format:
Zhenlin Wu, Haosong Zhao, Hongyuan Liu, Wujie Wen, and Jiajia Li. 2025. gHyPart: GPU-friendly End-to-End Hypergraph Partitioner. *ACM Trans. Arch. Code Optim.* 22, 1, Article 38 (March 2025), 25 pages. https://doi.org/10.1145/3711925

New Paper, Not an Extension of a Conference Paper.
This material is based upon work supported by the National Natural Science Foundation of China (#62302416) and the Guangzhou-HKUST(GZ) Joint Funding Program (#2023A03J0138).
Authors' Contact Information: Zhenlin Wu, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, Guangdong, China; e-mail: zwu065@connect.hkust-gz.edu.cn; Haosong Zhao, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, Guangdong, China; e-mail: hzhao037@connect.hkust-gz.edu.cn; Hongyuan Liu (Corresponding author), The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, Guangdong, China; e-mail: hongyuanliu@hkust-gz.edu.cn; Wujie Wen, North Carolina State University at Raleigh, Raleigh, North Carolina, United States; e-mail: wwen2@ncsu.edu; Jiajia Li, North Carolina State University at Raleigh, Raleigh, North Carolina, United States; e-mail: jiajia.li@ncsu.edu.

ACM 1544-3973/2025/03-ART38
https://doi.org/10.1145/3711925

1 Introduction

A hypergraph is a generalization of a graph in which a generalized edge can connect any number of nodes. Hypergraph partitioning is usually used to divide the hypergraph models into manageable components for various objectives, such as balancing loads and minimizing interconnect delays. Hypergraph partitioning has been widely adopted in numerous fields such as high-performance computing [42, 56, 63], scientific computing [8, 11, 25], (hyper-)graph analytics and learning [20, 22, 24], tensor network contraction [17, 50, 62], SAT solving [59], and VLSI design [9, 13].

In recent years, incorporating parallel computing to accelerate VLSI design stages, such as placement, timing analysis, and routing, is becoming increasingly popular [18, 34, 35, 38]. In the partitioning stage, timely results are increasingly crucial for VLSI designers seeking iterative improvements as chip complexity grows. Existing partitioners [16, 39] parallelize the partitioning algorithm on multi-core CPUs, but performance gains are limited by the number of CPU threads. Although some works [3, 4] have focused on optimizing graph partitioning on GPUs, GPU-accelerated hypergraph partitioning remains unexplored. Large hypergraphs, with a substantial number of hyperedges, could benefit from the massive data parallelism and compute power of GPUs compared with multi-core CPUs (e.g., NVIDIA RTX 3090 GPU has 10496 CUDA cores). This advancement could benefit numerous application domains that rely on efficient hypergraph partitioning. For instance, to enhance performance, **Sparse Matrix-Vector Multiplication** (**SpMV**) often requires value reordering. However, this process can be costly, typically relying on hypergraph partitioning [42, 56]. Therefore, adopting a low-overhead hypergraph partitioning approach can significantly boost SpMV performance. Also, hypergraph partitioning tools can be utilized in finding efficient contraction paths [17, 50] in tensor network contraction. The parallelism hierarchy of GPUs (grids, thread blocks, and threads) with affiliated synchronization strategies at each level expands optimization opportunities by supporting more parallelization strategies for different GPU routines in a hypergraph partitioning algorithm. Besides, the memory-intensive nature of the algorithm can benefit from the high bandwidth offered by GPUs.

To unleash the performance of GPUs for hypergraph partitioning, we first propose a baseline implementation, gHyPart-B. This end-to-end GPU hypergraph partitioner is a reasonable representative work as we follow the mainstream *multi-level* partitioning framework [6, 10, 13] for its high-quality and fast runtime. It adapts BiPart [39], a deterministic multi-threaded CPU-based hypergraph partitioner, to GPUs. However, we cannot fully exploit the performance potentials by directly porting it to GPUs. Naïve GPU designs such as gHyPart-B face performance issues due to the following challenges:

Challenge #1: Thread Underutilization. We observe that assigning one thread to process a hyperedge in existing CPU partitioning algorithms [16, 39] is inefficient on GPUs. GPU threads are underutilized since the workload assigned to each thread highly depends on the characteristics of the hyperedges of input hypergraphs. While the **Cooperative Thread Array** (**CTA**) scheduler can perform coarse-grained load balancing [26], fine-grained parallelism is needed to efficiently utilize GPU threads.

Challenge #2: Long Critical Path. We found that the existing hypergraph partitioner suffers from long critical path.[1] Hypergraph partitioners employing the multi-level paradigm consist of three phases: coarsening, initial partitioning, and refinement. A subroutine of the coarsening phase, *hypergraph construction*, generates new data structures of a coarsened hypergraph. A *linear search* operation applied in BiPart aims to generate new hyperedges without duplicate nodes, but introduces redundant workload and long critical path [19], resulting in an $O(|V|^2)$ per thread complexity. This further exacerbates thread underutilization and becomes a bottleneck issue.

[1] "Critical path" refers to the longest series of sequential operations in a parallel computation.

Table 1. Comparison of Our Work with Previous Works

	Sequential	Multithread	GPU	Deterministic	Parallelizations
hMetis [12]	✔				
PaToH [64]	✔				
Zoltan [25]		✔			Fixed
BiPart [39]		✔		✔	Fixed
Mt-KaHyPar [16]		✔		✔	Fixed
Our work			✔	✔	Adaptive

Challenge #3: No One-Size-Fits-All Implementation. A hypergraph partitioner calls many GPU kernels and deals with diverse input hypergraphs. Consequently, the execution characteristics such as parallelism, data locality, thread utilization, and critical path could be different among kernel algorithms and all depend on the features of input hypergraphs. Thus, choosing the most suitable combination of GPU kernels for a hypergraph partitioner is challenging, especially when seeking optimal performance on diverse input data.

To address the three challenges, we propose an end-to-end optimization solution, collectively named gHyPart. gHyPart contains two components: the first addresses **Challenges #1 and #2** by careful redesigning GPU kernels with fine-grained parallelization strategies and reduced computational complexity, while the second tackles **Challenge #3** by proposing a decision tree-based strategy that selects the most suitable combination of kernel implementations for an input hypergraph. The decision tree is trained via using feature-preserved *synthetic hypergraphs*, thereby ensuring the model's portability and also eliminating the need for accessing real-world data that could be proprietary. Compared with prior works in Table 1, to the best of our knowledge, this is the *first* end-to-end deterministic hypergraph partitioner on GPUs.

In summary, this work[2] makes the following contributions:

- We present the first high-performance end-to-end GPU-based hypergraph partitioning implementation. Our detailed characterization reveals three major performance challenges. (Section 3)
- We recognize parallelization strategies as crucial in GPU kernel's performance for different hypergraphs. To take advantage of this, we redesign critical kernels to support diverse strategies and maintain determinism, substantially expanding the optimization space. (Sections 3.3 and 4)
- A lightweight strategy selection mechanism is introduced to identify the most suitable parallelization strategy for each subroutine and input hypergraph, thereby enhancing overall performance. (Section 5)
- A detailed evaluation across 500 real-world hypergraphs demonstrates that the proposed approach outperforms BiPart [39], Mt-KaHyPar-SDet [16] and gHyPart-B with speedups of up to 125.7× (17.5× on average), 640.0× (24.2× on average), and 171.6× (1.4× on average), respectively, while maintaining comparable partitioning quality to these works. (Sections 6 and 7)

2 Preliminaries

Hypergraph. Hypergraphs are generalizations of the graph models where each hyperedge can connect any number of *nodes* in H. We denote a hypergraph $H = (V, E)$, where V is the set of nodes and E stands for the set of hyperedges (*hedges* in short). A hyperedge connects multiple vertices

[2] gHyPart is open-source on GitHub at https://github.com/zlwu92/gHyPart_TACO

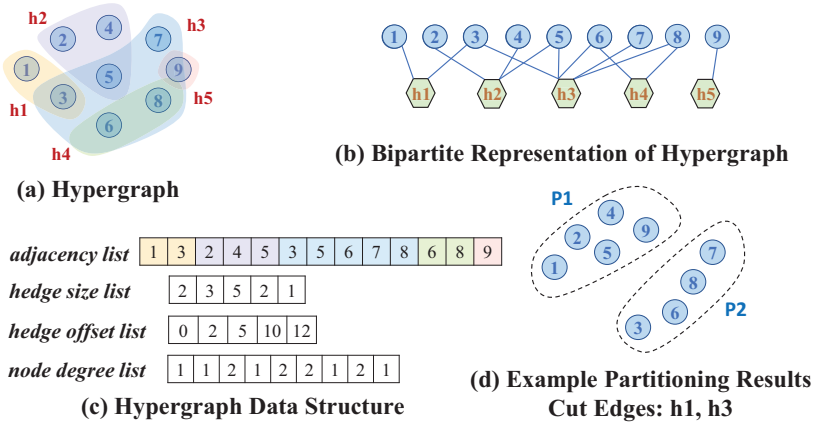


Fig. 1. A hypergraph (a) could be represented as a bipartite graph (b). The corresponding memory layout (c), and an example of partitioning the hypergraph (d).

in a hypergraph, generalizing edges in graphs. Nodes or hyperedges in many hypergraphs have numerical attributes called weights. Our focus is on unweighted hypergraphs, where all edges and vertices have a uniform weight of one. For example, Figure 1(a) shows an example hypergraph that has 5 hyperedges and 9 nodes.

Storage of Hypergraph. Hypergraphs can be converted into bipartite graphs for efficient memory storage. In Figure 1(a), h1 connects node 1 and node 3, resulting in edges connecting h1 to node 1 and node 3 in the bipartite graph in Figure 1(b). Memory storage for hypergraphs, shown in Figure 1(c), includes: (1) `adjacency list` storing node IDs for each hyperedge, (2) `hedge size list` tracking hyperedge sizes, (3) `hedge offset list` recording starting indices for each hyperedge, and (4) `node degree list` storing node degrees representing the number of incident hyperedges.

Hypergraph Partitioning Problem. Given a hypergraph $H = (V, E)$, and an imbalance parameter $\epsilon \in (0, 1)$, the 2-way hypergraph partitioning problem is to partition the node set into two subsets V_1 and V_2, such that V_1 and V_2 satisfy $|V_i| \leq (1+\epsilon)\lceil |V|/2 \rceil$. Generally, finding a ϵ-balanced k-way partition is to obtain $V_1, V_2, \ldots, V_k$, where each V_i is a subset of the *node set* V and $V_1 \cup \cdots \cup V_k = V$, where each V_i is a set of nodes that satisfy $|V_i| \leq (1+\epsilon)\lceil |V|/k \rceil$ (balance constraint). The goal of hypergraph partitioning is to minimize connectivity metric (i.e., hyperedge cut) $cut(H, P) = \sum_{e \in E}(\lambda(e) - 1)$, where $\lambda(e) := |V_i | V_i \cap e = \varnothing|$ represents the number of different sets connected by hyperedge e under the balance constraint. In Figure 1(d), the original hypergraph is partitioned into two partitions, labeled P1 and P2. Hyperedges h1 and h3 span across both partitions, resulting in them being "cut" by the partitioning. This results in $\lambda(h1) = \lambda(h3) = 2$. Therefore, the total number of edge cuts should be calculated as follows: $(2-1) + (2-1) = 2$.

Multi-level Hypergraph Partitioning Framework. While some works employ embedding [28], spectral [44, 52, 53], or other partitioning techniques [21, 30], most use the "multi-level" approach [6, 9, 13] to solve large-scale partitioning problems. As Figure 2 shows, the general workflow of this approach involves three phases: (1) *coarsening*, (2) *initial partitioning*, and (3) *refinement*. The coarsening phase reduces the problem scale iteratively by merging nodes into supernodes likely to be assigned to the same set, generating a sequence of progressively coarsened and smaller hypergraphs. This process continues until the hypergraph size meets a predefined threshold. Initial partitioning involves bisection of the coarsest hypergraph, while refinement improves bipartition

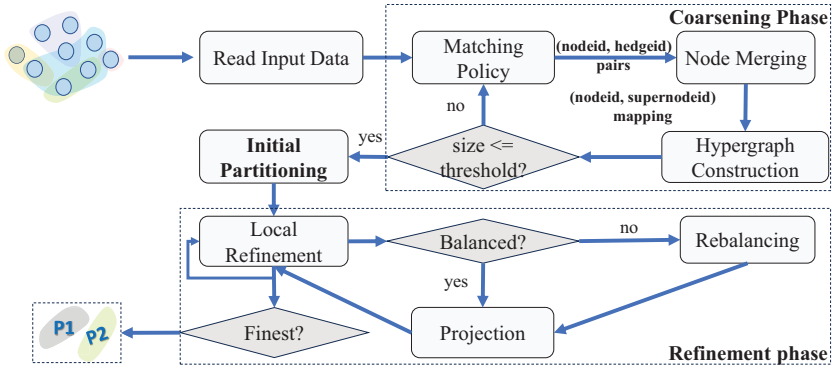


Fig. 2. Overview of hypergraph partitioning workflow.

quality through iterative local search and node swapping heuristics to ensure the balance requirement [39, 65]. The refined results are then projected back to the original hypergraph.

3 Characterization and Motivation

This section introduces the baseline end-to-end GPU implementation and outlines performance challenges to efficiently utilize GPU resources via detailed characterization. Section 3.3 introduces the parallelization strategies we utilize in our solution.

3.1 gHyPart-B: Baseline End-to-End Hypergraph Partitioning on GPUs

To understand the performance characteristics of hypergraph partitioning on GPUs, we adapt an existing deterministic multi-thread partitioner, BiPart [39], for GPUs with minimal changes to the original algorithm. We choose BiPart [39], which is known for these properties. Given the shared functionalities of multi-level partitioners, our approach can extend to other algorithms.

Our GPU Baseline: gHyPart-B. Our baseline, gHyPart-B, uses a fixed parallelization scheme following BiPart for each compute kernel. It covers all three phases: coarsening (*one-hyperedge-per-thread*), initial partitioning, and refinement (*one-node-per-thread*), with 22 GPU kernels as shown in Table 2. The coarsening phase involves (node, hedge) matching, node merging, and hypergraph construction. Initial partitioning quickly divides hypergraphs into two sets, while the refinement phase optimizes partitioning quality by swapping nodes to maintain balance. Besides, gHyPart-B inherits BiPart's deterministic algorithms, employing tie-breaking with the smallest assigned IDs and fixed execution order.

Expensive Procedures: Node Merging and Hypergraph Construction. We have evaluated 100 hypergraphs with the longest execution time from 500 real-world datasets described in Section 6 using our gHyPart-B implementation. We concentrate our attention on GPU kernels because our observations suggest that memory allocations and copy operations make only a negligible contribution to the overall execution time, accounting for approximately 3% of the total execution time (geometric mean) for our evaluated hypergraphs. The time percentage results are presented in Figure 3 and we find that for larger hypergraphs, core procedures (matching, node merging, construction) in the coarsening phase together take more than 70% of the total GPU kernel time. Consequently, this motivates us to examine the GPU efficiency of kernels emphasizing on these specific procedures in Section 3.2. In this work, we do not emphasize on further optimizations for the refinement phase as it uses a loop of tiny steps (`K16` ~ `K21` in Table 2), instead of some heavy procedures, to maintain balance constraints during node movements in BiPart's algorithm. Future work will explore algorithmic enhancements for the performance and quality of this phase.

Table 2. Kernel Information in gHyPart-B

Phases	Parallel Pattern Info. / Kernel Info.	
Coarsening	(node, hedge) matching	P1 K0: assign_hedge_priority
		P1 K1: assign_node_first_priority
		P1 K2: assign_node_second_priority
		P1 K3: assign_node_hedge_pairs
	Node Merging	P1 K4: node_merging_I
		P1 K5: node_merging_II
		P1 K6: mark_kept_hedges
		K7: self_merging
		K8: coarse_node_renumbering
	Hypergraph Construction	P2 K9: mark_unique_coarse_nodes
		P1 K10: add_coarse_nodes_to_adjacent_list
		K11: set_coarse_node_properties
Initial Partitioning	P1 K12: setup_initial_partitioning	
	K13: compute_nodes_move_gain	
	K14: sort_nodes_by_ratio_of_gain_to_weight	
	K15: apply_initial_move	
Refinement	P1 K16: compute_nodes_move_gain	
	K17: sort_nodes_by_gain_foreach_partition	
	K18: node_swapping_move	
	P1 K19: compute_nodes_move_gain (rebalancing)	
	K20: sort_nodes_by_ratio_of_gain_to_weight	
	K21: apply_node_move	

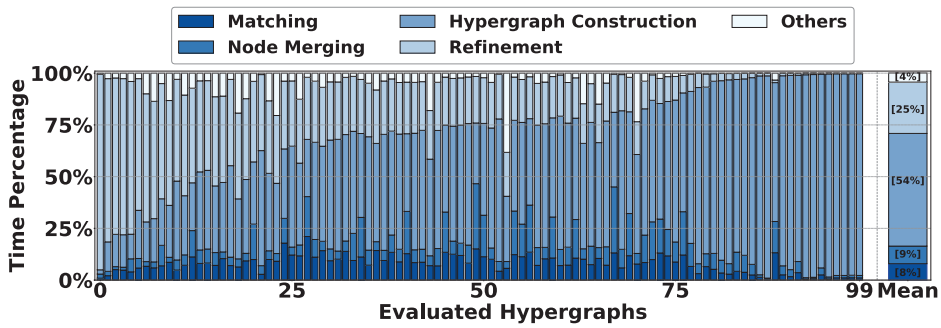


Fig. 3. Breakdown of time spent on each procedure of gHyPart-B on the 100 hypergraphs with the longest execution time.

3.2 Analysis of GPU Kernel Efficiency

We identify two challenges when applying the same parallelization approach in BiPart, to gHyPart-B due to the SIMT execution model of GPUs.

Challenge #1: Thread Underutilization. *The one-hyperedge-per-thread parallelization results in poor kernel efficiency on GPUs due to thread underutilization.* In this approach, the workload assigned to each thread is determined by the hyperedge size. When different-sized hyperedges are assigned to threads in a warp, the execution time of the entire warp is influenced by the largest workload, resulting in idle threads within the warp. We quantitatively measure its impact using the

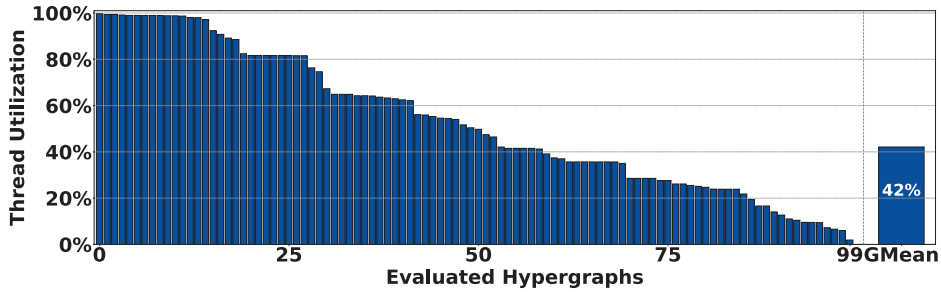


Fig. 4. The thread utilization of gHyPart-B on the 100 hypergraphs with the longest execution time.

following methodology: when a thread accesses a node within a hyperedge, we consider it to have completed one unit of work. The *overall thread utilization* is defined as the geometric mean of *per warp utilization* across all warps which is formally described in Equations (2) and (1) ($N = \#Warps$).

$$Overall\ Thread\ Utilization = \left(\prod_{i=1}^{N} Warp_i\ Utilization\right)^{\frac{1}{N}}, \tag{1}$$

$$Warp\ Utilization = \frac{\#\ Useful\ Work}{\#\ Total\ Work}. \tag{2}$$

Taking the example of dac12 hypergraph from physical design benchmark suite [43], with a maximum hyperedge size of 8,488 and a minimum hyperedge size of 1, our metric indicates an overall utilization of 41%. Figure 4 shows thread utilization results for various input hypergraphs. In terms of geometric mean, the utilization of the GPU threads is achieved only 42% in the 100 evaluated hypergraphs, suggesting *suboptimal efficiency of the kernels in gHyPart-B*. Therefore, maximizing warp utilization is essential for achieving high efficiency and performance on GPUs.

Challenge #2: Long Critical Path and Suboptimal Workload. *gHyPart-B suffers from a higher workload and longer critical path in the hypergraph construction procedure, which becomes a performance bottleneck on GPUs.* Figure 3 demonstrates that *hypergraph construction*, a subroutine called in every iteration of the coarsening phase, takes an average of 54% of GPU kernel time in gHyPart-B. This procedure constructs the coarsened hypergraph's data structures using a node-to-supernode mapping. During the addition of supernodes to the coarsened hypergraph's adjacency list, a thread performs a *linear search* to avoid duplicate entries. In BiPart's design, the node order in the hyperedge's adjacency list affects the coarsening results in the next iteration, thereby influencing the final partitioning results. Thus, using linear search is a valid approach to achieve both objectives: (1) generate adjacency list without duplicate nodes; (2) the generated node order is deterministic. This maintains the determinism of the partitioning results. For example, if node IDs 1, 2, and 3 are mapped to supernode IDs 1, 2, and 1, respectively, then supernode IDs 1, and 2 are added to the coarsened hypergraph. When the procedure encounters node ID 3, a linear search on the supernodes of all its neighbors confirms that its supernode (1) has already been added, preventing the addition of another supernode at this point. However, such a linear search not only exacerbates the thread underutilization problem but also brings unnecessary workload and critical path [19] in gHyPart-B as we port BiPart's design to GPU: each thread has a $O(|V|^2)$ critical path, and the total amount of workload of all threads is $O(|V|^2|E|)$. The "critical path" is a significant performance metric in parallel algorithms, representing the longest sequence of sequential operations in a parallel computation. The *depth* [5], which corresponds to the length of this critical path, represents the optimal running time achievable on an ideal machine with an unlimited number of processors. In other words, a long critical path indicates *severe load imbalance* across threads.

These challenges motivate us to *expand the optimization space for various hypergraphs*, as further described in Sections 3.3 and 4.

3.3 Parallelization Strategies

The challenges demonstrated in Section 3.2 motivate us to apply GPU-friendly parallelization strategies to the hypergraph partitioner and adapt them to input hypergraphs.

Parallelization strategies. In this work, we have utilized three primary parallelization strategies and adopted the PRAM model to analyze their performance tradeoffs. Since the total work coming from the original algorithm does not change, we order them by the length of the critical path (i.e., *depth* [19, 23]) of their computations: P1 (**one-hyperedge-per-thread**), commonly used in prior works and our baseline (gHyPart-B), schedules one thread to process the whole adjacency list (refer to Figure 1(c)) of a hyperedge; P2 (**one-hyperedge-per-thread-block**), arranges all threads of one thread block to process the whole adjacency list of a hyperedge, thus threads could access contiguous memory addresses; P3 (**one-adjacent-entry-per-thread**), assigns one thread to process each node in the adjacency list, ensuring even workload distribution. A parallelization strategy with a shorter critical path generally suggests a more balanced workload distribution among the threads, leading to improved thread utilization and memory bandwidth utilization. However, this can increase synchronization granularity, decrease data reuse, and lower cache efficiency. From the parallelism granularity perspective, P1 is coarse-grained, P3 is fine-grained, and P2 is intermediate in granularity.

4 Kernel Redesign with Different Parallelization Strategies

In this section, we tackle **Challenges #1 and #2** by optimizing critical kernels with the aforementioned parallelization strategies in Section 3.3, while ensuring the correctness and determinism. We present redesigned GPU kernels for the time-consuming node merging and hypergraph construction procedures. The matching kernels, with a shorter execution time, are not detailed here as their redesigns are straightforward. Our proposed approaches, along with their corresponding parallelization strategies, are alternative choices that can be independently selected. This expansion of choices provides a broader optimization design space which will be discussed in Section 5.

4.1 Node Merging

Workflow. Figure 5 depicts the node merging workflow [39], comprising three steps: *node merging I*, *node merging II*, and *self merging*. *Node merging I* kernel relies on output from a "multi-node matching"[3] algorithm [39], resulting in a many-to-one mapping between node IDs and hyperedge IDs. Each node in a hyperedge has a (nodeID, hedgeID) pair. Nodes matched to the same hyperedge will eventually be merged together into one node. Then, both *node merging II* and *self merging* are designed to merge the remaining unmerged nodes from *node merging I*. Node merging generates supernodes from nodes, with the supernode ID determined by the lowest node ID for determinism.

Redesigning Node Merging with Kernel Splitting. To simplify the discussion, we describe *node merging I* kernel as an example. The gHyPart-B's implementation with P1 sequentially performs condition checking to determine if the node matches the hyperedge, weight accumulation, and generating mapping to supernodes. We use the same pre-defined parameter, *weight limit*. This constraint is crucial to preventing the excessive merging of nodes into a single supernode and

[3]If all included nodes of a hyperedge are merged into one node during the later node merging procedure, this hyperedge can be eliminated in the next level coarsened hypergraph. This method is designed to achieve both hyperedge reduction and node reduction during each iteration of the multilevel coarsening, thereby accelerating coarsening processing. Please refer to BiPart [39] for more details.

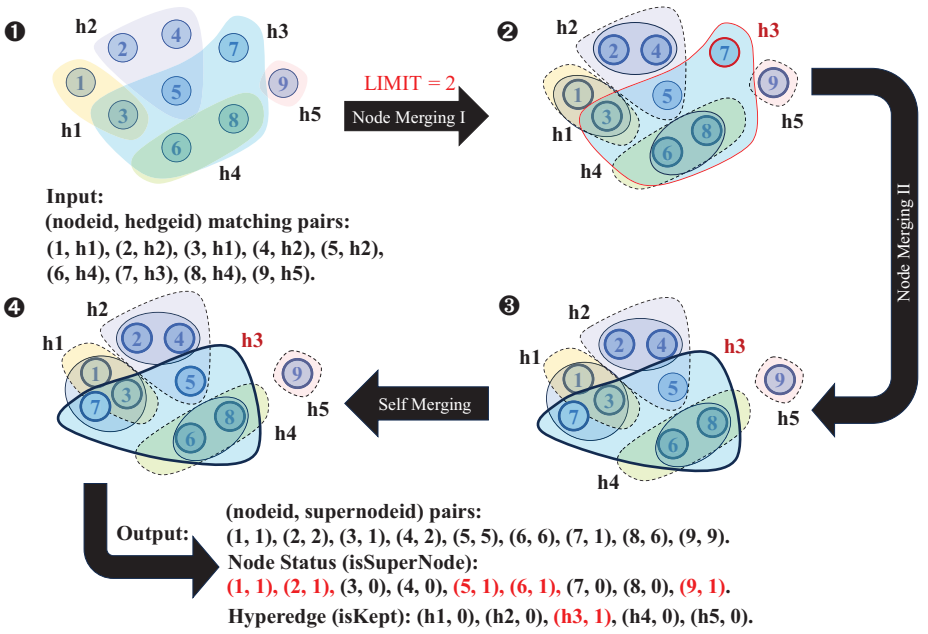


Fig. 5. Illustrating the node merging procedure.

achieving both coarsening efficiency and balancing. In Figure 5 ❶, nodes 2 and 4 are merged into a supernode, but node 5 is not merged with them due to the weight limit (2).

To enhance GPU thread utilization, we redesign the kernels with fine-grained parallelization strategies. However, simultaneous work by multiple threads in checking and selecting merging nodes within each hyperedge can result in different node orders, breaking determinism in BiPart's algorithm. In contrast, P1 assigns only one thread per hyperedge, ensuring consistent node ordering but sacrificing GPU utilization. To achieve both determinism and improve parallelism, we split the process into four sub-kernels that can utilize P2 and P3 without compromising determinism. Figure 6 illustrates our implementation where each thread is responsible for a node in the adjacency lists of hyperedges (i.e., P3). Besides, a thread must know which hyperedge it is working on. This is done by maintaining an auxiliary list `hyperedge_ID` with the same length as the adjacency list.

Algorithm 1 illustrates how we redesign the kernels with the given example in Figure 6. The first GPU kernel (Figure 6 ❶) prepares a `weight_list` storing the weight of each node in the adjacency list (in this example, each node weight is 1). Each thread i is tasked with determining if the node in the adjacency list corresponds to the hyperedge it is currently processing (`hyperedge_ID[i]`) . If it is a match (see (nodeID, hedgeID) pairs in Figure 5 ❶), the thread writes the associated weight to the `weight_list[i]`; otherwise, it writes 0 to the same location. Second (Figure 6 ❷), we perform an inclusive scan (`thrust::inclusive_scan`) over the `weight_list` and compute a `local_weight_list` showing the accumulated weights (***segmented prefix sum***) for each hyperedge. Third (Figure 6 ❸), we need to determine which node should be merged based on the `local_weight_list`. To search all neighbor locations of the `local_weight_list` where the accumulated weight fits within the weight limit in a deterministic manner, we adopt a different approach: we calculate the differences between adjacent elements in the `local_weight_list`. Subsequently, starting from the leftmost neighbor of each hyperedge, we determine that each node j should be mapped to the supernode only if the calculated difference is non-zero, and the `local_weight_list[j]` does not exceed the weight limit. Meanwhile, we compute the *lowest ID*

ALGORITHM 1: Redesigned "Node Merging I" Procedure with P3

```
   // Figure 6 ❶
1  Function subkernel_func1():
2      for len(Adjs) ← 0 to tid in thread-parallel do
3          // Check if node matches to current hyperedge
4          if hedgeIDs[tid] == pairs[tid] then
5              // Mark its weight for further weight condition checking
6              weights[tid] = 1;
7          end
8      end
9  End
   // Figure 6 ❷
10 Function subkernel_func2():
11     // Perform device-wide inclusive scan on weight_list
12     thrust::inclusive_scan(thrust::device, weights, weights+len(weights),
        weights);
13 End
   // Figure 6 ❸
14 Function subkernel_func3():
15     for len(Adjs) ← 0 to tid in thread-parallel do
16         // Get the local accumulated weight within each hyperedge
17         local_offset = hedgeOffs[hedgeIDs[tid]];
18         local_weight = weight[tid] − weight[local_offset];
           // Calculate weight diff between adjacent nodes
19         diff = weights[tid] − weights[tid − 1];
20         // Check if current accumulated weight is within the limit threshold for
              this hyperedge
21         if diff > 0 && local_weight <= LIMIT then
22             nodeid = Adjs[tid];
23             isCandidate[nodeid] = 1;
24             supernodes[hedgeIDs[tid]] = atomicMin(nodeid, supernodes[hedgeIDs[tid]]);
25         end
26     end
27 End
   // Figure 6 ❹
28 Function subkernel_func4():
29     for len(Adjs) ← 0 to tid in thread-parallel do
30         // Update node-to-supernode mapping
31         if isCandidate[Adjs[tid]] then
32             supernode = supernodes[Adjs[tid]];
33             node_mapping[Adjs[tid]] = supernode;
34             atomicAdd(&node_weights[supernode], node_weights[Adjs[tid]]);
35         end
36     end
37 End
```

among each of these selected nodes as their supernode (e.g., node 2 for h2). Ultimately (Figure 6 ❹), we generate the mapping from nodes to supernodes and update the node weight in the next iteration of the coarsening phase. The redesigned kernels with P3 are also deterministic because they compute the lowest IDs to break ties via atomic operations.

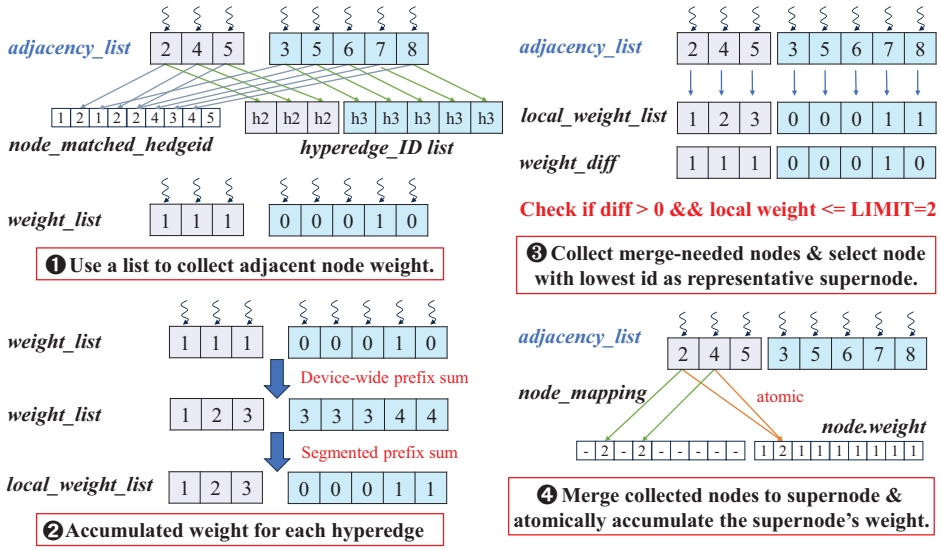


Fig. 6. Redesigned "node merge I" kernel (K4) with P3.

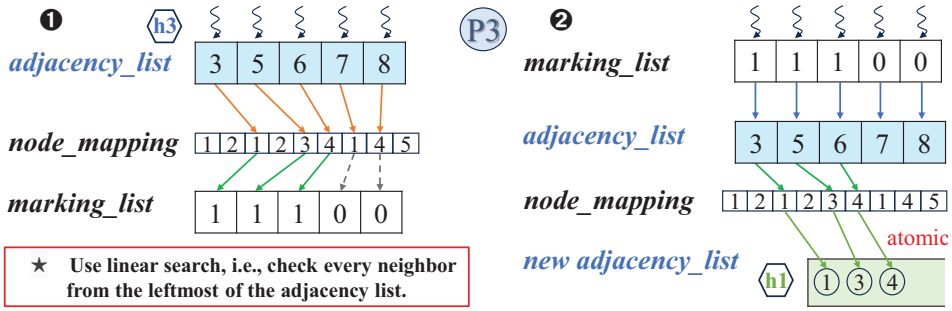
In addition to the demonstrated "node merging I" step, we also redesign kernels for "node merging II" and "multi-node matching" procedures with both P3 and P2 using a similar kernel splitting strategy to enhance the thread utilization.

4.2 Hypergraph Construction

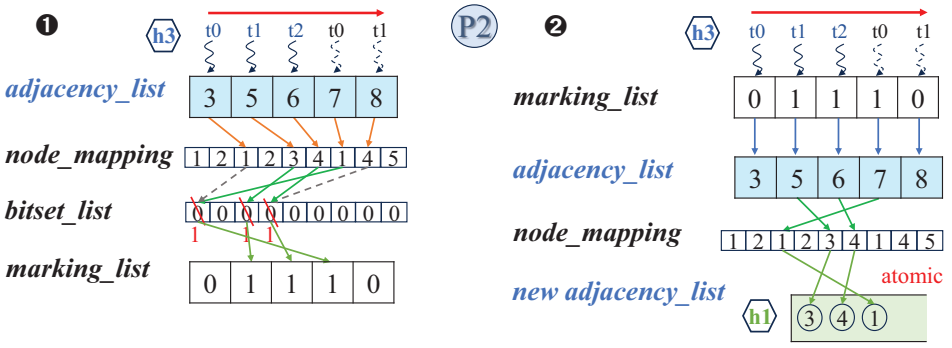
Overview. During coarsening, each iteration creates a new hypergraph from a finer one to reduce the problem scale. This procedure takes the output of node merging (discussed in Section 4.1), which is a mapping from node ID to supernode ID showing which supernodes the nodes are merged into, and a hyperedge status list showing which hyperedges are kept (Figure 5). Hypergraph construction produces hyperedge offset, node degree, and adjacency lists. Deduplication is required before adding `node_mapping[i]` to the adjacency list due to many-to-one mapping. In gHyPart-B, this step leads to a $O(|V|^2)$ per thread complexity via using linear search. We proposed optimizations for this process in two parallelization strategies:

Improving Thread Utilization. Leveraging P3, our *first* approach assigns the nodes of hyperedge adjacency lists to individual threads, effectively reducing the critical path of computations and maximizing thread utilization. Similar to the redesign for node merging kernels with P3, we also require the `hyperedge_ID` to access the hyperedge index information. In Figure 7(a) ❶, the `node_mapping` array maps node IDs to supernode IDs. Each thread i checks whether `node_mapping[i]` equals to `node_mapping[j]` (i.e., linear search) by looking up every node j from the beginning of the adjacency list to $i-1$ to avoid generating duplication in the adjacency list of the coarsened hypergraph (Figure 7(a) ★). The `marking_list[i]` is marked 1 if thread i does not find any `node_mapping[j]` equals to `node_mapping[i]`. After updating the `marking_list`, Figure 7(a) ❷ demonstrates that each thread i writes to the new `adjacency_list` of the coarsened hypergraph if `marking_list[i]` is 1. Meanwhile, we will generate the next level `hyperedge_ID` that will be used in the next iteration. In summary, this approach ensures that the critical path for each thread is lowered to $O(|V|)$, while the overall workload remains bounded at $O(|V|^2|E|)$.

Reducing Critical Path. In our *second* approach, where one thread block is still mapped to each hyperedge, as illustrated in Figure 7(b) ❶, each block of threads must iteratively fetch the neighbors in the adjacency list and check their mapping supernode. To reduce the work complexity



(a) Redesigned kernels with P3.



(b) Redesigned kernels with bitset-based P2 (P2b).

Fig. 7. Redesigned hypergraph construction kernels.

and streamline the process, we opt to maintain a bitset (`bitset_list`) for each hyperedge in this approach. We refer to this method as "P2b" in the rest of our article. `bitset_list[`i`]` is set if at least one thread intends to fetch a supernode i via accessing the `node_mapping` list. By examining the extracted bit value from the specific location of the `bitset_list[`i`]`, we can determine whether the corresponding supernode has been accessed and updated by a thread for the first time . Therefore, `marking_list` is updated by each thread according to the results of setting the bitset followed by a thread block synchronization (`__syncthreads()`). Figure 7(b) ❷ creates the adjacency lists of the coarsened hypergraph similar as Figure 7(a) ❷. In summary, the adoption of a bitset-based approach with P2 yields several advantages, such as a critical path reduction to $O(1)$, a total workload decrease to $O(|V||E|)$, and an improved utilization of threads.

Determinism. Note that either approach may yield different supernode IDs, causing nondeterminism in the new adjacency list order. For example, in Figure 7, the first approach orders nodes as 1→3→4, while the second orders as 3→4→1. To ensure determinism, we apply a *stable* ***segmented sort*** (`cub::DeviceSegmentedSort::SortKeys`) [47] on adjacency lists, where each hyperedge's list is treated as a segment, ensuring deterministic neighbor ordering. While introducing overhead, this approach's benefits outweigh the cost (8.5% on average).

5 Parallel Strategy Selection with Hypergraph Characteristics

In Section 4, we redesign kernels for node matching, node merging, and hypergraph construction. This results in $3 \times 3 \times 3 = 27$ possible execution paths for a given input hypergraph. Traversing all paths is time-consuming, thus a selection strategy is born out of necessity.

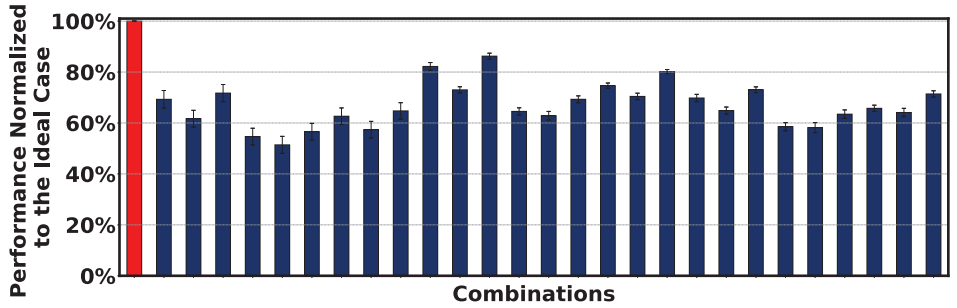


Fig. 8. Performance of each fixed strategy combination normalized to the ideal case. Evaluation Methodology in Section 6.

Challenge #3: No One-Size-Fits-All Implementation. We evaluate strategy selection's impact across hypergraphs, establishing an "ideal" reference case and testing other combinations. Figure 8 illustrates results. Two observations emerge: (1) No single combination consistently matches the ideal performance for all hypergraphs; (2) Even the best-performing fixed combination (P3-P1-P3) falls short of the ideal, highlighting the need for a dynamic selection mechanism. In conclusion, superior performance across diverse hypergraphs requires a carefully designed strategy selection scheme.

Tradeoffs and Challenges in Parallelization Strategy Selection. Selecting an efficient parallelization strategy for a given hypergraph is challenging due to numerous tradeoffs in execution. Take hyperedge size distribution as an example, while employing the parallelization strategy P1, where each thread processes a hyperedge, if the hyperedge sizes demonstrate a skewed distribution, the GPU threads are underutilized due to the imbalanced workload. In contrast, the redesigned kernels employing P2 or P3 improve thread utilization when dealing with skewed hypergraphs. However, to achieve this enhancement, P2 and P3 require kernel splitting (discussed in Section 4.1), which hampers temporal locality. Thus, it is difficult to devise a selection strategy that accounts for the characteristics of the input hypergraph, the available parallel patterns, and the varying capabilities of GPUs.

Machine Learning-based Modeling. Selecting the most appropriate strategy combination based on the input hypergraph features can be transformed into a traditional machine learning (ML) classification problem. In this way, the hypergraph characteristics serve as input X, and the selected strategy is the predicted output y. However, we must meet *two requirements* for our end-to-end GPU hypergraph partitioner: (1) In the field of VLSI design, obtaining real datasets may be challenging [40]. Users may lack sufficient real hypergraphs as training data. In such cases, directly using real hypergraphs for training may not be feasible. (2) The runtime prediction must be lightweight to minimize overhead and interpretable to facilitate insights for future optimizations.

Methodology. To meet the *first* requirement, we synthesize hypergraphs to train our classifier. We generate 500 synthetic hypergraphs. To ensure that the synthetic hypergraphs closely resemble real ones, we synthesize each hypergraph based on a real hypergraph's characteristics, for example, making them have the same number of hyperedges, number of nodes, and *hyperedge size distribution*. To introduce diversity, we generate the hypernode IDs randomly for each hyperedge to provide various connectivity structures. To meet the *second* requirement, we train decision tree classifiers to select strategies for the three subroutines (node matching, node merging, and hypergraph construction). We selected the decision tree model as it is lightweight and demonstrates interpretability as its capability of ranking feature importance. To train the classifiers, the first column of Table 3 displays a set of 11 selected hypergraph characteristics.

Table 3. Characterstics of 500 Evaluated Hypergraphs by Percentile

Features	Min	25%	50%	75%	Max
$\|V\|$	7,502	33,737	112,904	560,124	13,378,617
$\|E\|$	163	38,662	142,228	849,856	13,378,617
AdjListSize	6,323	304,467	1,509,696	4,266,705	283,073,458
AvgHESize	1	3	6	14	1,825
SdHESize	0.0	1.0	4.2	21.7	3,500.8
MinHESize	1	1	1	3	126
MaxHESize	3	11	60	664	395,259
AvgHNDeg	1	4	7	16	1,260
SdHNDeg	0.0	2.3	8.5	25.8	3,500.8
MinHNDeg	0	1	1	3	126
MaxHNDeg	2	23	128	1,061	395,259

"AdjListSize" stands for the total number of nodes stored in the adjacency list. "HN" and "HE" stand for hypernode and hyperedge, respectively. "Sd" stands for standard deviation.

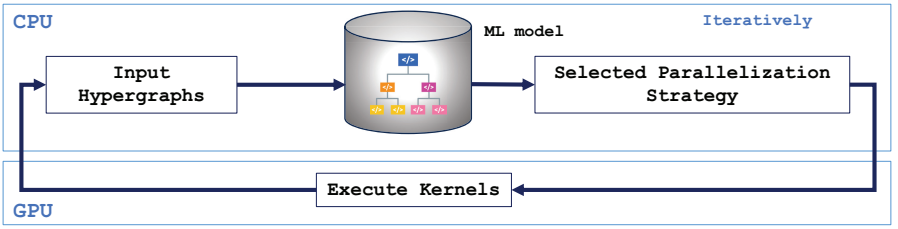


Fig. 9. Overview of the iterative runtime decision-making workflow.

We employ the characteristics of *synthetic hypergraphs* as the offline training input (`trainX`) and incorporate optimal parallelization strategies as the training target output (`trainY`) to develop the decision tree model, thus ensuring no data leakage from the training set to the testing set. During runtime, the actual hypergraph characteristics are used as testing input (X), and the selected parallelization strategy is the expected output (y). We set the maximum depth to 2 to mitigate overfitting, while all other parameters are configured to their default values in the `DecisionTreeClassifier()` API provided by the Python `scikit-learn` package.

Workflow of Decision Making. Our implementation deploys the decision tree on the CPU side. Figure 9 presents the overall workflow of our decision tree-based iterative selection strategy integrated in gHyPart. During each iteration of the coarsening stage, the CPU retrieves the characteristics of the current hypergraph and uses them as input for the decision tree model. The model then predicts optimal parallelization strategies. Subsequently, the corresponding GPU kernels are launched for the partitioning task.

Trained Model. Figure 10 shows our selection scheme based on trained decision trees. The decision tree for the node matching procedure is excluded since it has no leaf whose size is larger than the required minimal leaf size, except for P3. This implies that P3 will be consistently chosen for our redesigned matching kernels in our selection scheme. This strategy is employed with our redesigned kernels throughout the workflow, enabling runtime adaptability for selecting the most suitable strategy with our optimizations.

6 Evaluation Methodology

System Configurations. We perform experiments on an NVIDIA RTX 3090 GPU (Ampere architecture [45], 24 GB memory, 6 MB L2 cache, and 82 SMs) hosted by Intel Xeon 4214R CPU

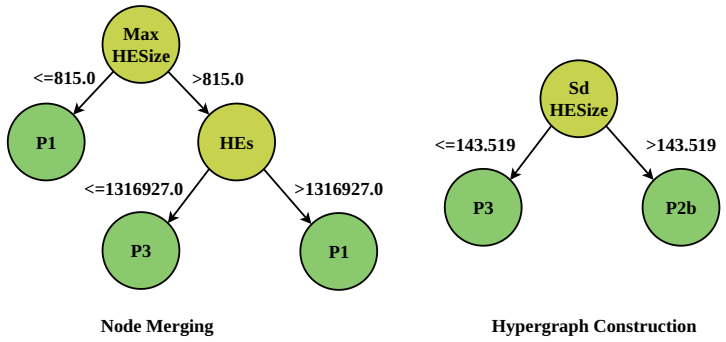


Fig. 10. Decision tree classifiers for node merging and hypergraph construction.

Table 4. Evaluated Schemes

Scheme	Description
BiPart [39]	Multi-thread CPU execution with 12 threads.
Mt-KaHyPar [16]	Multi-thread CPU execution with 12 threads.
hMetis [12]	Sequential high-quality partitioner.
gHyPart-B	Our GPU implementation with fixed default parallel strategies.
gHyPart-P2	Our GPU implementation with fixed P2.
gHyPart-P3	Our GPU implementation with fixed P3.
gHyPart-R	Our GPU implementation with randomly selected strategies.
gHyPart	Our GPU implementation with adaptively selected parallelization strategies.
gHyPart-O	Our GPU implementation with oracular parallel strategies.

(12 cores), 128 GB memory, and Linux operating system. gHyPart is written in CUDA/C++, compiled using the `-O3 -arch=sm_86` flag with GCC 9.4 and CUDA 12.0. We use an NVIDIA RTX 4060Ti (Ada Lovelace architecture [46], 8 GB memory, 32 MB L2 cache, and 34 SMs) for GPU sensitivity study.

Evaluated Hypergraphs. Our dataset [29, 49] comprises 500 hypergraphs collected from existing partitioning works [16, 39]. Hypergraphs are in hMetis format [12]. Instances from SuiteSparse Matrix Collection [57] are transformed into hypergraphs using row-net model [63]. Each row represents a hyperedge, and columns represent nodes. Additional hypergraphs are from ISPD98 VLSI Circuit Benchmark Suite [1], DAC 2012 Routability-Driven Placement Contest [43], and 2014 SAT Competition [2]. Table 3 summarizes percentile information of the hypergraph characteristics.

Partitioning Configurations. In our experiments, we set the imbalance ratio ϵ to 0.05, a commonly used value in the CPU comparative works [12, 16, 39]. Moreover, we assess the bipartition performance on all 500 real-world hypergraphs for each of the evaluated schemes in Table 4, with a specific focus on the GPU kernel performance. We also evaluate the performance comparison for different k-way partitioning tasks. For hMetis, we set a timeout constraint to 5× of the BiPart's longest execution time for all hypergraphs. Additionally, we use 128 as the block size for all kernels since we have observed that the overall performance is not sensitive to the block sizes.

Evaluated Schemes. Table 4 presents the evaluated schemes in our experiment. We evaluate two state-of-the-art multi-thread CPU works: BiPart [39] and Mt-KaHyPar [16], both recognized for their high quality, high performance, and deterministic nature. These two CPU works are executed with 12 threads. We also compare our works with hMetis [12], a sequential CPU partitioner. Additionally, we evaluate six GPU-based schemes: gHyPart-B, gHyPart-P2, gHyPart-P3, gHyPart-R, gHyPart, and gHyPart-O. gHyPart-B, gHyPart-P2, and gHyPart-P3 use a fixed parallelization

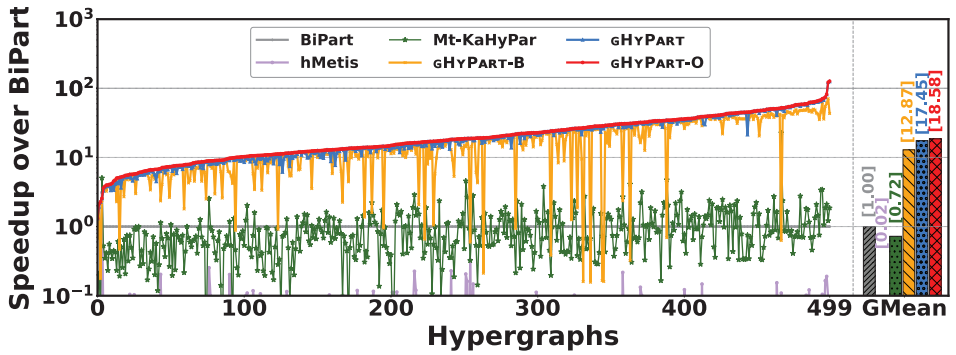


Fig. 11. End-to-end speedup over BiPart.

strategy. gHyPart-R employs a random selection strategy for the three procedures. gHyPart is the primary solution in this work, featuring redesigned kernels with our lightweight kernel selection strategy. gHyPart-O selects the combination of strategies that yields the best end-to-end performance through a grid search of all possible combinations of different kernel implementations and the corresponding parallelization strategies. Here, gHyPart-O is the reference case "ideal" in Section 5.

7 Experimental Results

7.1 End-to-End Performance

We first assess gHyPart's efficiency across 500 hypergraphs for 2-way partitioning, comparing it to BiPart and Mt-KaHyPar-SDet. Results in Figure 11 are sorted by speedup of gHyPart-O over BiPart in ascending order. We make the following observations: First, gHyPart-B achieves 12.9× average speedup over BiPart. Second, gHyPart, the primary solution in this work, features redesigned kernels with our lightweight kernel selection strategy. On average, it achieves 17.5× speedup across the evaluated hypergraphs. Third, gHyPart achieves an approximate 17.45/0.72 = 24.2× and 17.45/0.02 = 872.5× average speedup over Mt-KaHyPar and hMetis on 4214R CPU, respectively. Lastly, compared with gHyPart-O's 18.6×, gHyPart closely approaches, while gHyPart-B shows a significant and fluctuating performance gap.

Figure 12 evaluates the effectiveness of gHyPart on the 100 hypergraphs in Figure 3. It is observed that gHyPart achieves up to a 171.6× speedup and an average of 3.1× speedup over gHyPart-B. This indicates that careful redesign of the kernel algorithms, combined with a lightweight kernel selection strategy, significantly boosts GPU performance, especially for large hypergraphs.

We further evaluate gHyPart across different benchmark suites. According to Figure 13, our gHyPart achieves significant speedup for all the evaluated benchmark suites. We observe that gHyPart performs better in DAC2012 [43] and SAT2014 [2]. The key reason is that our tailored strategies integrated within gHyPart are more effective for hypergraphs with large sizes and power-law distributions in hyperedge and node quantities.

7.2 Kernel Speedup with Different Strategies

Next, we evaluate the performance of node merging and hypergraph construction kernels with different parallelization strategies.

Node Merging. Figure 14(a) illustrates the achieved speedup for node merging kernels using P2 and P3 compared to the baseline implementation employing P1. Although the pie chart in

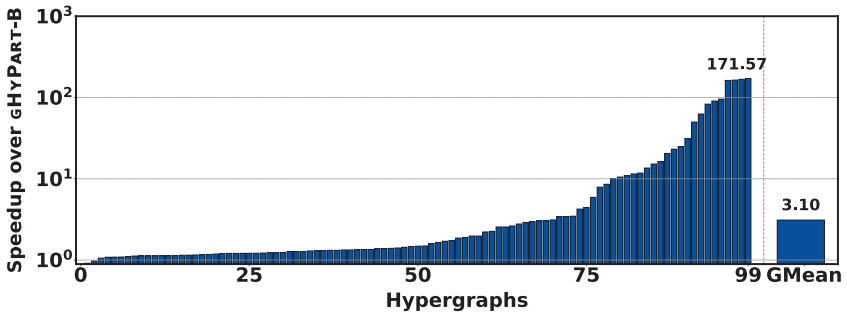


Fig. 12. Speedup of gHyPart over gHyPart-B in the 100 most time-consuming hypergraphs. Results are sorted by the speedup in ascending order.

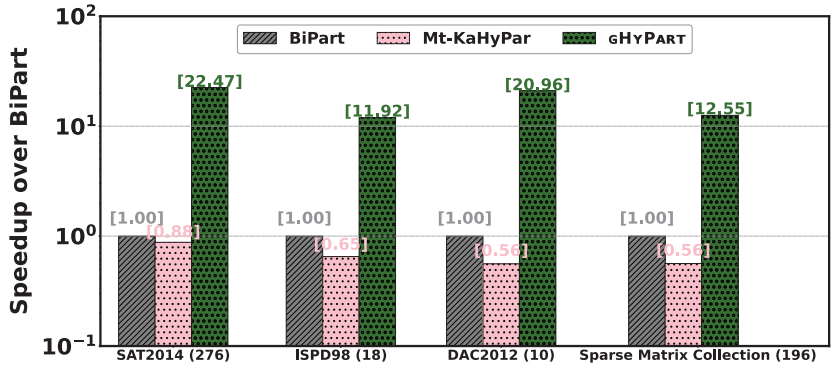


Fig. 13. Overall speedup relative to BiPart across benchmark suites. The number of hypergraphs for each benchmark suite is specified in parentheses.

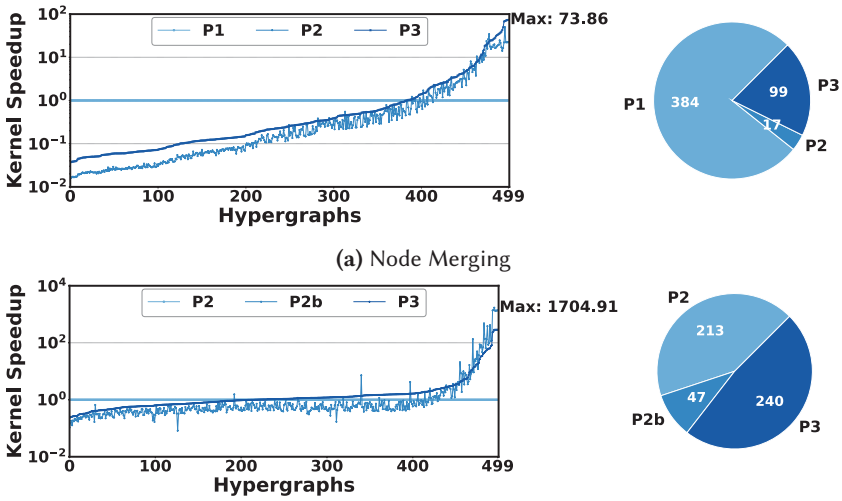
Figure 14(a) indicates that the baseline implementation with P1 performs the best on more than 75% of the evaluated hypergraphs, our redesigned kernels with P2 and P3 outperform the baseline kernel on over 100 instances. Additionally, P3 achieves a maximum speedup of 73.9×, particularly benefiting hypergraphs with an imbalanced distribution of hyperedges.

Hypergraph Construction. Figure 14(b) presents the efficiency of our redesigned construction kernels employing P3 and P2 ("P2b") in comparison to the baseline implementation which incorporates P2. The right-hand pie chart reveals that the optimized kernels with P3 outperform others in nearly half of the evaluated hypergraphs. According to the line chart in Figure 14(b), the "P2b" approach attains a maximal speedup exceeding 1700.0× compared to the baseline.

Overall, both Figure 14(a) and 14(b) suggest that no single parallelization strategy consistently delivers superior performance across all hypergraphs, underscoring the importance of a judicious selection strategy.

7.3 Selection Strategy

To demonstrate the effectiveness of our selection strategy based on a decision tree model, Figure 15 shows the performance normalized to gHyPart-O for all evaluated hypergraphs. On average, gHyPart achieves approximately 94% of the performance attained by gHyPart-O. In contrast, gHyPart-B, gHyPart-P2, and gHyPart-P3, which utilize a fixed parallelization strategy for the three procedures, achieve 69.3%, 58.2%, and 80.2% of gHyPart-O's performance, respectively. gHyPart-R, which randomly selects strategies for kernels in each procedure, achieves only 60.0%



(a) Node Merging

(b) Hypergraph Construction

Fig. 14. **Left:** Kernel speedup using various parallelization strategies over baseline implementation across evaluated hypergraphs. Results are sorted by the speedup of P3 in ascending order. **Right:** # hypergraphs achieving optimal performance with each strategy.

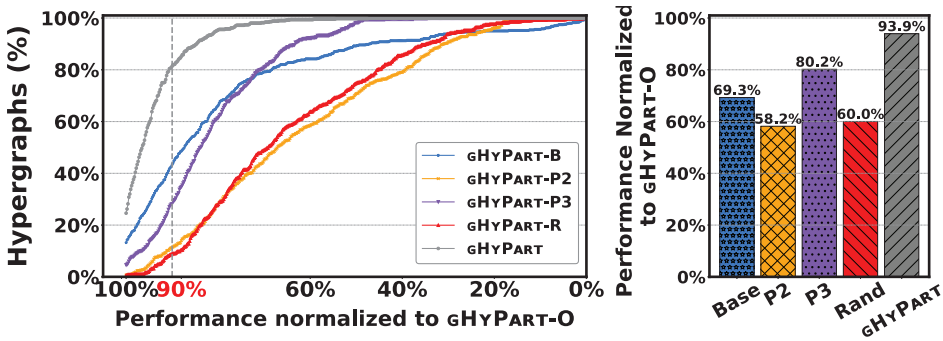


Fig. 15. **Left:** Performance comparison for our GPU implementations with different selection strategies on RTX 3090 GPU. **Right:** "Base" stands for gHyPart-B, "P2" stands for gHyPart-P2, "P3" stands for gHyPart-P3, "Rand" stands for gHyPart-R, and "gHyPart" is our scheme.

of gHyPart-O's performance. In summary, gHyPart consistently outperforms other alternative strategies, achieving over 90% of gHyPart-O's performance for more than 80% of all the evaluated hypergraphs (as shown on the left side of Figure 15). This demonstrates the effectiveness of our proposed selection strategy in maximizing performance across a wide range of hypergraphs .

7.4 Sensitivity to GPU Architectures

We conduct performance sensitivity studies to evaluate how gHyPart behaves on a different RTX 4060Ti GPU:

Performance. We have evaluated our schemes on an RTX 4060Ti GPU and compared it with BiPart executed on the same CPU. The left chart of Figure 16 illustrates that gHyPart on RTX 4060Ti achieves an average speedup of 15.5× over BiPart, indicating strong performance on

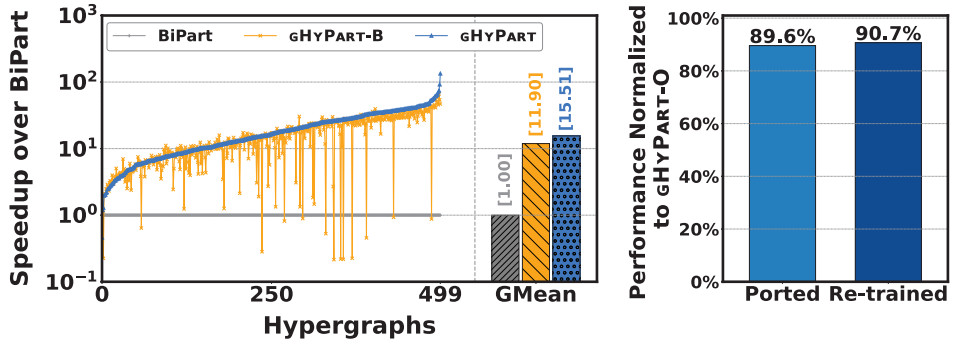


Fig. 16. Sensitivity to RTX 4060Ti GPU. **Left:** End-to-end speedup over BiPart on RTX 4060Ti GPU. **Right:** "Ported" implies utilizing model trained for RTX 3090 to RTX 4060Ti, and "Re-trained" represents gHyPart with model re-trained for RTX 4060Ti.

Table 5. CPUs Used in the Sensitivity Study

CPU specs	Xeon Silver 4214R	I9-13900K	Xeon Platinum 8358P
Core Count	12	24	32
Frequency (GHz)	2.4	3.0	2.6
LLC Size (MB)	16.5	36	48

alternative GPUs. However, the speedup achieved on the RTX 4060Ti is lower than that on the RTX 3090 (17.5×), primarily due to the RTX 3090's greater number of Streaming Multiprocessors (SMs), which facilitates a higher degree of parallelism. Therefore, strategies focusing on maximizing the utilization of parallelization resources are more apt to achieve improved computational efficiency and attain superior performance outcomes.

Selection Strategy. The comparative analysis presented in the right chart of Figure 16 provides valuable insights into the performance of our selection model when applied to different GPU architectures. Specifically, we evaluate the model's performance when ported from an RTX 3090 GPU and retrained on an RTX 4060Ti GPU. We make two observations: First, retraining the model on the new target gains performance. Second, the model ported from the RTX 3090 performs nearly as well as the model retrained on the RTX 4060Ti GPU, with only a marginal 1.1% performance decrease. This observation highlights the robustness and transferability of our proposed selection strategy. Overall, both observations demonstrate that our proposed selection strategy is not only simple but also effective, showcasing its portability across different GPU architectures.

7.5 Sensitivity to Different CPUs

We introduced two extra CPUs for running BiPart and Mt-KaHyPar: a desktop processor (I9-13900K) and a high-end server processor (Xeon 8358P). Table 5 presents the specifications of evaluated CPUs. Figure 17 shows the overall performance speedup over BiPart's performance on 4214R CPU.

We make the following observations: First, on I9-13900K CPU, both BiPart and Mt-KaHyPar achieve 1.66× and 1.57× speedup on average compared to their performance on Xeon 4214R CPU, respectively. Second, on the 8358P CPU, BiPart and Mt-KaHyPar attain speedups of 0.9× and 1.4×, respectively, compared with the 4214R CPU. Third, our gHyPart can still achieve an average of

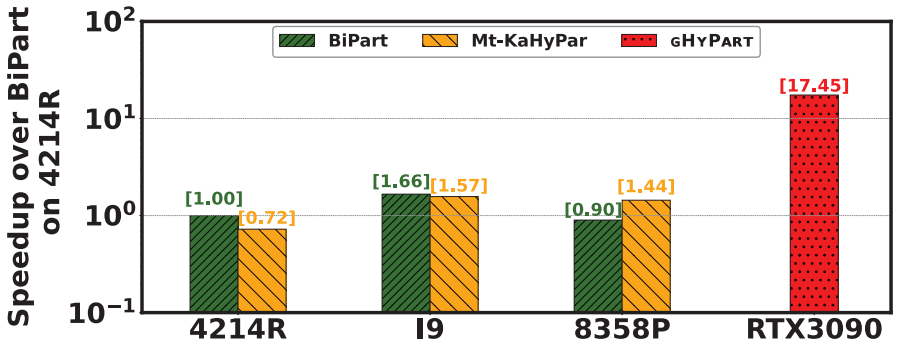


Fig. 17. Performance sensitivity to different CPUs.

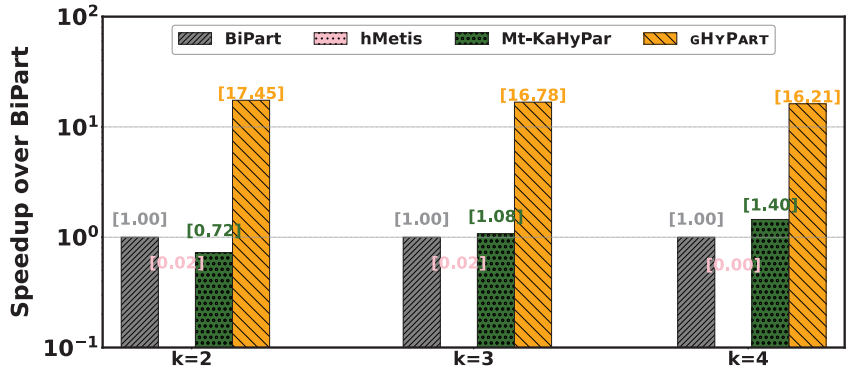


Fig. 18. End-to-end speedup over BiPart for k-way partitioning tasks.

17.45/1.66 = 10.5× and 17.45/1.57 = 11.1× speedup over BiPart and Mt-KaHyPar on the I9-13900K CPU with 24 threads, respectively.

Based on the observations, we make the following analysis. First, BiPart and Mt-KaHyPar show superior performance on I9-13900K CPU due to its larger LLC per thread and higher peak performance, benefiting irregular applications. Second, they underperform on the high-end server CPU; BiPart's performance on the 8358P CPU is 90% of that on the 4214R CPU, and Mt-KaHyPar lags behind the I9 desktop CPU, indicating scalability challenges. Overall, our gHyPart delivers notable gains over these CPU-based solutions.

7.6 Evaluation of k-Way Partitioning

We evaluate the performance of gHyPart on k-way partitioning tasks. Our k-way partitioning method follows a recursive bisection approach similar to BiPart [39]. Figure 18 shows the average end-to-end speedup of gHyPart compared with BiPart for $k = 2, 3, 4$. We make the following observations. First, gHyPart achieves speedups of 16.8× and 16.2× over BiPart for $k = 3$ and $k = 4$, respectively. Second, the performance of Mt-KaHyPar improves relative to BiPart as k increases, since BiPart requires an additional rebalancing step in its recursive bisection process for k-way partitioning. Third, the overall speedup of gHyPart for k-way ($k > 2$) partitioning is slightly lower than that for 2-way partitioning, as larger values of k reduce the data parallelism available for GPUs on the evaluated hypergraphs. Fourth, hMetis performs extremely slowly. Thus, we set a timeout for each hypergraph at 5× the longest execution time of BiPart. On average, gHyPart

Table 6. Partitioning Quality Results

Work A vs. Work B	k = 2 (Total 385 for hMetis)			k = 3 (Total 422 for hMetis)			k = 4 (Total 469 for hMetis)		
	% Better	% Comparable	% Worse	% Better	% Comparable	% Worse	% Better	% Comparable	% Worse
gHyPart vs. Bipart	6%	91%	3%	11%	80%	9%	13%	78%	9%
gHyPart vs. Mt-KaHyPar	6%	71%	23%	1%	34%	65%	2%	28%	70%
gHyPart vs. hMetis	18%	50%	32%	16%	21%	63%	16%	23%	61%

hMetis processes a subset of the evaluated hypergraphs within the specified timeout.

attains a more than 872.5× speedup over hMetis for different values of k. Overall, gHyPart exhibits impressive performance against existing CPU works.

7.7 Partitioning Quality

To mitigate the impact of edge-cut differences in some large hypergraphs on the overall evaluation, we employ the ratio of the edge-cut difference to the total number of hyperedges. This reasonable metric allows us to effectively compare the partitioning quality between the two methods. Specifically, we calculate the difference in edge cuts produced by method A and a comparative reference method B relative to the total number of hyperedges in a given hypergraph $H = (V, E)$. This metric can be expressed as the ratio $(cut_A(H) - cut_B(H))/|E|$. A negative ratio signifies that method A outperforms method B, whereas a positive ratio indicates that method B performs better. A close-to-zero ratio suggests that both methods achieve similar partitioning quality. This approach ensures a more fair comparison by accounting for the varying sizes and complexities of different hypergraphs.

Our method categorizes results into "Better", "Comparable", and "Worse" based on the ratio value. Specifically, if the ratio's absolute value is less than 0.5%, it indicates that our approach achieves a partitioning quality similar to that of counterpart methods. The classifications for the other two categories are determined in a similar manner.

As illustrated in Table 6, the quality of gHyPart aligns closely with BiPart. Since hMetis is prohibitively slow for processing all hypergraphs, we report quality results only for those hypergraphs that hMetis could complete within the timeout. Although the overall quality of Mt-KaHyPar and hMetis is better than gHyPart for k-way (especially when $k > 2$) partitioning, this is mainly because our algorithm follows BiPart's design (recursive bipartitioning) not the direct k-way partitioning in Mt-KaHyPar and hMetis. Our work still obtains substantial performance improvement for $k > 2$ (exceeding 1000× over hMetis and 16.21/1.40 = 11.6× over Mt-KaHyPar) cases; the performance acceleration approaches could inspire the GPU design of the direct k-way partitioning method.

7.8 Comparison with HyperG

HyperG [31] is a recent GPU-based hypergraph partitioning work. Since HyperG is not open-source and evaluated on only 18 hypergraphs, we compare gHyPart with it using the same set of hypergraphs. Our experiments demonstrate an average 140× speedup under identical configurations ($k = 2$, $\epsilon = 0.03$) and achieve better partitioning quality in half of these benchmarks.

8 Related Work

In this section, we discuss some related work from three distinct perspectives: hypergraph partitioners implemented on CPUs, graph partitioning optimization techniques utilized on GPUs, and various other GPU-accelerated graph applications. By examining these areas, we provide a comprehensive overview of the current advancements and methodologies in the field.

Hypergraph Partitioners on CPUs. hMetis [13] and PaToH [63] are the earliest CPU-based hypergraph partitioning tools. KaHyPar [51] and its derivatives [50, 58, 61] enhance sequential partitioning quality. Zoltan [25] is designed for optimizing large-scale distributed sparse matrix-based computations (SpMV/SpMM). BiPart [39] and Mt-KaHyPar-SDet [16] (a deterministic version of Mt-KaHyPar [15, 29]) ensure deterministic results. TritonPart [7] is a recent hypergraph partitioning framework specifically designed for diverse constraint-driven tasks in VLSI domains. MedPart [33] follows the multi-level paradigm and develops a fast spectral coarsening method. *However, all these partitioning tools are implemented on CPUs and could not exploit and leverage fine-grained parallelizations for diverse hypergraphs.*

GPU-Accelerated (Hyper-)Graph Partitioning. Cheng et al. [36] present a GPU-based work focusing on a different coarsening algorithm instead of an end-to-end hypergraph partitioner. Goodarzi et al. [4] propose a multilevel graph partitioning framework that leverages both GPU and CPU for different phases of the process, showcasing a hybrid approach to address the complexities of graph partitioning tasks. Their subsequent research [3] targets enhancing intra-warp load balancing problems in graph partitioning. Gilbert et al. [14] propose a GPU parallelization of the heavy-edge coarsening method designed for multilevel graph partitioning. Additionally, G-kway [32] is a recent multilevel GPU-accelerated graph partitioner. However, directly applying graph-based methods to hypergraph partitioning can lead to a reduction in parallelism due to fundamental differences between graphs and hypergraphs that a hyperedge can link to any number of nodes. *Therefore, it is important to note that these GPU partitioning works are still unable to accomplish an end-to-end partitioning workflow for hypergraphs.* HyperG [31] is a recent study aimed at enhancing hypergraph partitioning on GPUs through enhanced parallelism and better utilization of GPU threads. *Compared to this work, our work has developed different parallelization strategies on GPUs to provide a comprehensive understanding of the performance and achieved performance gains across different types of hypergraphs with a runtime selection strategy. Furthermore, our partitioner is deterministic and achieves this with low overhead.*

Parallelizations for Graph Applications. Existing works [54, 55] optimize graph processing frameworks on various architectures with diverse parallelization strategies, considering the irregularity in memory accesses and parallelism. For GPUs, existing works [37, 41, 48] implement hierarchical load balance thread mapping schemes in multiple granularities. GSwitch [27] and SEP-Graph [60] leverage runtime information and graph characteristics. Hypergraph partitioning, unlike graph processing, relies on heuristic algorithms, posing challenges for fixed strategies. Thus, directly using existing approaches is not realistic to create a fast end-to-end hypergraph petitioner on GPUs. *Our work proposes kernel redesign and a strategy selection mechanism to improve performance and efficiency within the hypergraph partitioning workflow.*

9 Conclusion

We introduce gHyPart, the first high-performance deterministic end-to-end hypergraph partitioner on GPUs. To address challenges stemming from diverse hypergraph characteristics and fixed parallelizations, we expand the optimization space by redesigning partitioning algorithms with various parallelization strategies. A lightweight strategy selection approach, guided by a machine learning model, is utilized to achieve optimal performance. This approach enables the selection of the most suitable parallelization strategies based on runtime hypergraph characteristics. Our comprehensive evaluation demonstrates that gHyPart significantly outperforms existing state-of-the-art hypergraph partitioners while maintaining comparable partitioning quality for the majority of hypergraphs. Our future work aims to enhance gHyPart by exploiting and integrating more efficient generic heuristic algorithms, targeting improved partition quality and GPU performance.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback, which significantly improved the quality of the paper. We thank Le Qin and Dr. Jiayi Huang's group for providing the I9-13900K desktop computer used in CPU evaluation.

References

[1] Charles J. Alpert. 1998. The ISPD98 circuit benchmark suite. In *Proceedings of the International Symposium on Physical Design (ISPD'98)*.

[2] Anton Belov, Daniel Diepold, Marijn J. H. Heule, and Matti Järvisalo. 2014. The 2014 sat competition benchmarks. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*.

[3] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High performance multilevel graph partitioning on GPU. In *International Conference on High Performance Computing & Simulation (HPCS'19)*.

[4] Bahareh Goodarzi, Martin Burtscher, and Dhrubajyoti Goswami. 2016. Parallel graph partitioning on a CPU-GPU architecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16)*.

[5] Guy E. Blelloch. 1996. Programming parallel algorithms. *Communication of the ACM* 39, 3 (1996), 85–97.

[6] Bruce Hendrickson and Robert Leland. 1995. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC'95)*.

[7] Ismail Bustany, Grigor Gasparyan, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. 2023. An open-source constraints-driven general partitioning multi-tool for VLSI physical design. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD'23)*.

[8] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert Heaphy, and Lee Ann Riesen. 2007. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*.

[9] Jong-Sheng Cherng and Sao-Jie Chen. 2003. An efficient multi-level partitioning algorithm for VLSI circuits. In *Proceedings of the 16th International Conference on VLSI Design (VLSI'03)*.

[10] Jason Cong and M'Lissa Smith. 1993. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *Proceedings of the 30th International Design Automation Conference (DAC'93)*.

[11] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*.

[12] George Karypis and Vipin Kumar. 1998. hMetis: A hypergraph partitioning package, version 1.5.3. In *Technical Mannual, Department of Computer Science, University of Minnesota*.

[13] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning applications in VLSI domain. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

[14] Michael S. Gilbert, Seher Acer, Erik G. Boman, Kamesh Madduri, and Sivasankaran Rajamanickam. 2021. Performance-portable graph coarsening for efficient multilevel graph analysis. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*.

[15] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2023. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms* 20, 1, Article No.: 9 (2023), 1–54.

[16] Lars Gottesbüren and Michael Hamann. 2022. Deterministic parallel hypergraph partitioning. In *28th International Conference on Parallel and Distributed Computing, Proceedings (Euro-Par'22)*.

[17] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (2021), 410.

[18] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated static timing analysis. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*.

[19] Guy E. Blelloch. 1996. Programming parallel algorithms. In *Communications of the ACM*.

[20] Aparajita Haldar. 2022. *Hypergraph-Based Optimisations for Scalable Graph Analytics and Learning*. Ph. D. Dissertation. University of Warwick.

[21] Hyunchul Shin and Chunghee Kim. 1993. A simple yet effective technique for partitioning. In *IEEE Transactions on Very Large Scale Integration Systems*.

[22] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. 2018. HyperX: A scalable hypergraph framework. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 5 (2018), 909–922.

[23] Joseph JáJá. 1992. An introduction to parallel algorithms. In *Addison-Wesley*.

[24] Julian Shun. 2020. Practical parallel hypergraph algorithms. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*.

[25] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. 2006. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*.

[26] Onur Kayıran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

[27] 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 (PPoPP'19)*.

[28] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. 2016. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining (WSDM'16)*.

[29] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Scalable shared-memory hypergraph partitioning. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX'21)*.

[30] Lars Hagen and Andrew B. Kahng. 1992. A new approach to effective circuit clustering. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'92)*.

[31] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way hypergraph partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC'25)*.

[32] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-accelerated k-way graph partitioner. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[33] Rongjian Liang, Anthony Agnesina, and Haoxing Ren. 2024. MedPart: A multi-level evolutionary differentiable hypergraph partitioner. In *Proceedings of the 2024 International Symposium on Physical Design*.

[34] Shiju Lin, Jinwei Liu, and Martin DF Wong. 2021. GAMER: GPU accelerated maze routing. In *Proceedings of the 40th IEEE/ACM International Conference on Computer-Aided Design (ICCAD'21)*.

[35] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2019. Dreamplace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement. In *Proceedings of the 56th Annual Design Automation Conference*. 2019, 1–6.

[36] Lin Cheng, Hyunsu Cho, and Peter Yoon. 2015. An accelerated procedure for hypergraph coarsening on the GPU. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'15)*.

[37] Hang Liu and H Howie Huang. 2019. SIMD-X: Programming and processing of graph algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*.

[38] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. 2013. NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (2013), 709–72.

[39] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. BiPart: A parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21)*.

[40] Maven Silicon. 2023. Introduction to Machine Learning in VLSI. Retrieved from https://www.maven-silicon.com/blog/machine-learning-in-vlsi/

[41] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. 117–128.

[42] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluc. 2024. Sparsity-aware communication for distributed graph neural network training. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP'24)*.

[43] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. 2012. The DAC 2012 routability-driven placement contest and benchmark suite. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*.

[44] Maxim Naumov and Timothy Moon. 2016. Parallel spectral graph partitioning. In *NVIDIA Technical Report*.

[45] NVIDIA Corporation. 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE. Retrieved from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

[46] NVIDIA Corporation. 2023. NVIDIA ADA GPU ARCHITECTURE. Retrieved from https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf

[47] NVIDIA Research. 2023. CUB: Cooperative primitives for CUDA C++. Release: 2.0.1.. In https://github.com/NVIDIA/cub

[48] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*.

[49] Sebastian Schlag. 2017. A benchmark set for multilevel hypergraph partitioning algorithms [Data set]. Zenodo. DOI: https://doi.org/10.5281/zenodo.291466
[50] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-quality hypergraph partitioning. In *ACM Journal of Experimental Algorithmics*.
[51] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. *k*-way hypergraph partitioning via *n*-Level recursive bisection. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*.
[52] Seher Acer, Erik G. Boman, and Sivasankaran Rajamanickam. 2020. SPHYNX: Spectral partitioning for HYbrid aNd aXelerator-enabled systems. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'20)*.
[53] Seher Acer, Erik G. Boman, Christian A. Glusa, and Sivasankaran Rajamanickam. 2021. Sphynx: A parallel multi-GPU graph partitioner for distributed-memory systems. In *Parallel Computing*.
[54] 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)*.
[55] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel local graph clustering. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1041–1052.
[56] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'15)*.
[57] Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. In *ACM Transactions on Mathematical Software (TOMS'11)*.
[58] Tobias Heuer and Sebastian Schlag. 2017. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *16th International Symposium on Experimental Algorithms (SEA'17)*.
[59] Vijay Durairaj and Priyank Kalla. 2004. Exploiting hypergraph partitioning for efficient boolean satisfiability. In *Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop*.
[60] 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)*.
[61] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a direct *k*-way hypergraph partitioning algorithm. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX'17)*.
[62] Fang Zhang. 2020. *A Parallel Tensor Network Contraction Algorithm and Its Applications in Quantum Computation*. Ph. D. Dissertation.
[63] Ümit V. Çatalyürek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. In *IEEE Transactions on Parallel and Distributed Systems (TPDS'99)*.
[64] Ümit V. Çatalyürek and Cevdet Aykanat. 1999. PaToH: Partitioning tool for hypergraphs. In *Technical Mannual*.
[65] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More recent advances in (Hyper)Graph partitioning. In *ACM Computing Surveys*.

Received 1 July 2024; revised 4 November 2024; accepted 25 December 2024