GPU-Accelerated Graph Partitioning Algorithms in VLSI Design

by

Wan Luan Lee

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 12/09/2025

Dedicated to my family, friends, and all those I have met along the way who have supported, encouraged, and believed in me.

ACKNOWLEDGMENTS

Completing this Ph.D. has been incredibly challenging, but it has also brought me profound personal and professional growth. Over these four years, I have thought about giving up many times, especially on those nights when I cried and had no idea how to move forward with my research. However, thanks to the support of the people who care about me, especially my advisor, committee members, colleagues, and family, I found the strength to continue and complete this journey. I could not have done it without their unwavering support.

First, I would like to express my deepest gratitude and appreciation to my Ph.D. advisor, Professor Tsung-Wei Huang. Professor Huang never gave up on me and continued to encourage me when I had almost lost confidence and faith in myself. I am truly grateful for his patience, guidance, and support. I could not have gone through this journey without him.

I also want to thank my committee members, Professor Umit Yusuf Ogras, Professor Joshua San Miguel, and Professor Xiangyao Yu. They gave me valuable feedback throughout my PhD studies. With their help, I made steady and positive progress in both my research and my development as a scholar.

I am grateful to my colleagues and friends, Dr. Dian Lun Lin, Dr. Jie Tong, Dr. Cheng Hsiang Chiu, Che Chang, Boyang Zhang, Chih-Chun Chang, Yi-Hua Chung, and Aditya Das Sarma for their encouragement and insightful discussions. They helped me develop and refine my research ideas and made my PhD journey more rewarding and enjoyable.

I also want to thank Bin Lin for his referral, which opened the door for me to join Cadence Design Systems. I am thankful to my hiring manager, Dongxiang Wu, for giving me an unforgettable opportunity to work as an intern and to continue in a full-time position, and to my mentor, Yen Shuo Chen, who helped me make strong progress in my internship work through many meaningful discussions. I am also grateful to everyone I met during my internship. They helped me grow as a researcher and engineer and encouraged me to bring my academic work into real industry applications.

I want to thank my family, especially my mother Judy Lee, my father Hector Lee, and my grandparents. They have always supported my decisions and believed that I would choose the right path. When I was burned out and overwhelmed by the pressure of research, they gently comforted me and reminded me that no matter what happens, home is always there for me, and they will always wait for me to return. Their love, trust, and constant understanding have given me the courage to keep moving forward and to complete this journey.

Moreover, I want to thank my significant other, Bingqi Shang, who supported me with patience, encouragement, and care in my everyday life. He listened to my worries, celebrated my progress, and reminded me to rest when I pushed myself too hard. I am deeply grateful for his love and companionship.

Finally, I want to thank my dog, April Lee, who moved with me from place to place and brought me comfort and happiness. Her presence made every new place feel more like home and helped me get through many difficult days with a smile.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Graph and hypergraph play critical roles in computer-aided design (CAD) because it allows us to break down a large circuit into several manageable pieces that facilitate efficient CAD algorithm designs. However, as circuit sizes continue to grow, partitioning becomes increasingly time-consuming. To address this runtime bottleneck, many researchers have leveraged CPU-parallel techniques to accelerate partitioning. However, the speedups are typically limited to 8–16 CPU threads. To overcome this challenge, this thesis leverages the massive parallelism of GPUs and introduces two GPU-accelerated partitioning algorithms: iG-kway for graphs and iHyperG for hypergraphs. G-kway features a union find-based coarsening algorithm that merges many vertices simultaneously, and a novel independent-set-based refinement algorithm that refines thousands of vertices in parallel. HyperG introduces a balanced group coarsening method and a sequence-based refinement algorithm. Experimental results show that G-kway achieves an average speedup of 8.6× over the CPU-parallel graph partitioner mt-metis [1], while HyperG delivers an average speedup of 4.1× over the CPU-parallel hypergraph partitioner Mt-KaHyPar [2], both maintaining comparable cut quality.

While G-kway and HyperG achieve new performance milestones in graph and hypergraph partitioning, they are limited to full partitioning. This lack of support for incrementality presents a critical limitation for many CAD applications, where circuits undergo iterative modifications as part of optimization loops. To address this limitation, this thesis also introduces two GPU-parallel incremental partitioning algorithms: iG-kway for graphs and iHyperG for hypergraphs. iG-kway features an incrementality-aware bucket-list data structure and a refinement kernel that refines only affected vertices. iHyperG introduces a scalable delta-based hypergraph data structure for efficient incremental modifications on the GPU, along with an effective incremental partitioning algorithm that rebalances the partition in a single pass and refines only cut-critical vertices. Experimental results show that iG-kway achieves an average speedup of 84× over the GPU-based full graph partitioner G-kway, while iHyperG delivers 190× speedup for hypergraph modification and 83× for partitioning over the state-of-the-art GPU-based full hypergraph partitioner, both maintaining comparable cut quality.

INTRODUCTION

Graph and hypergraph partitioning play important roles in various stages of computer-aided design (CAD), including placement, routing, timing analysis, and logic simulation [3]. For example, partitioning helps optimize component placement by dividing the circuit into smaller, more manageable blocks while minimizing interconnections between them [4]. Given that partitioning a graph or hypergraph from scratch (known as full partitioning) is NP-hard [5, 6, 2, 4], many heuristics have been developed [7, 8, 9]. Among them, multilevel partitioning is the most popular approach for large-scale graphs and hypergraphs due to its ability to generate high-quality partitions while ensuring fast runtime. A typical multilevel partitioner begins by iteratively coarsening the input graph or hypergraph into smaller representations. Once the structure becomes sufficiently small, the partitioner finds an initial partitioning result using a fast heuristic. It then performs uncoarsening, where the graph or hypergraph is iteratively restored to its original size. At each level of uncoarsening, a refinement algorithm is applied to improve partition quality.

However, as circuit sizes continue to grow, sequential full partitioners become increasingly time-consuming. For example, the sequential full graph partitioner Metis [10] can take five minutes to partition a five-million-gate circuit. Since partitioning can be performed multiple times during a CAD algorithm (e.g., incremental timing [11], RTL simulation [12]), the cumulative partitioning time can extend to several hours.

To address this challenge, existing full partitioners have leveraged multi-core CPUs to parallelize the partitioning process. For graph partitioning, mt-metis [1] is a state-of-the-art CPU-based partitioner that parallelizes the sequential k-way Fiduccia–Mattheyses (FM) algorithm [13]. For hypergraph partitioning, Mt-KaHyPar [2] leverages multithreading to parallelize both the coarsening and refinement algorithms, achieving significant speedups over the widely used sequential hypergraph partitioner hMetis [14]. Despite some improvements, the speedups of both mt-metis and Mt-KaHyPar typically plateau at 8 to 16 CPU threads [1, 2].

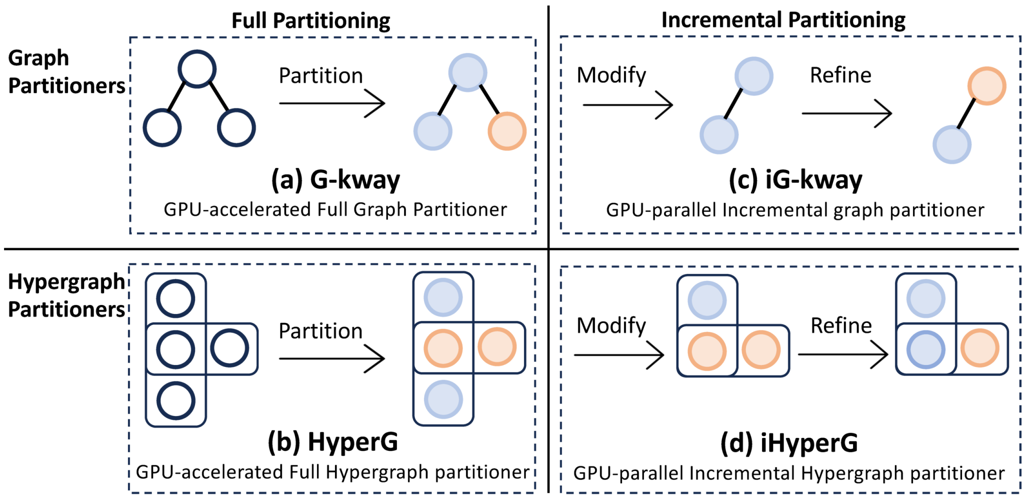

Full Partitioning
Incremental Partitioning
Graph Partitioners
Partition
Modify
Refine
(a) G-kway
GPU-accelerated Full Graph Partitioner
(c) iG-kway
GPU-parallel Incremental graph partitioner
Hypergraph Partitioners
Partition
Modify
Refine
(b) HyperG
GPU-accelerated Full Hypergraph partitioner
(d) iHyperG
GPU-parallel Incremental Hypergraph partitioner

Figure 0.1: The left two boxes show full graph (a) and hypergraph (b) partitioning, where the input is partitioned from scratch. The right two boxes show incremental graph (c) and hypergraph (d) partitioning, where the structure is first incrementally modified and then refined, saving significant runtime by avoiding repartitioning from scratch. In the graph partitioners, black lines represent edges connecting pairs of vertices. In the hypergraph partitioners, black boxes denote hyperedges, with their connected vertices enclosed inside the box.

To address this limitation, this thesis introduces two GPU-accelerated full partitioning algorithms that leverage the massive parallelism of modern GPUs to achieve acceleration beyond what is attainable with traditional CPU-parallel approaches. Figure 0.1 illustrates the workflows of (a) G-kway, a full graph partitioner, and (b) HyperG, a full hypergraph partitioner. Both algorithms partition the input from scratch and achieve significant speedups over widely used CPU-parallel graph and hypergraph partitioners. G-kway features a union find-based coarsening algorithm that merges many vertices simultaneously, and a novel independent-set-based refinement algorithm that refines thousands of vertices in parallel.

By exploiting massive parallelism in both coarsening and refinement, G-kway achieves significant speedup over CPU-parallel mt-metis [1] without compromising partitioning quality. On the other hand, HyperG redesigns both coarsening and refinement to handle the complex multi-pin relationships in hypergraphs by introducing a balanced group coarsening method and a sequence-based refinement algorithm, achieving substantial speedups over CPU-parallel Mt-KaHyPar [2] while preserving high partitioning quality.

While G-kway and HyperG achieve new performance milestones in graph and hypergraph partitioning by leveraging the massive parallelism of the GPU, they are limited to full partitioning, where the entire graph or hypergraph is partitioned from scratch. However, in many CAD applications that integrate partitioning into iterative optimization workflows, incremental partitioning can offer greater efficiency than full partitioning. For example, a timing optimizer may repeatedly adjust cell placements to meet timing goals [11], and logic synthesis tools may incrementally restructure logic cones to improve design quality [15]. In these iterative workflows, each time the circuit is incrementally modified, incremental partitioning can quickly refine the existing partitioning result to maintain reasonable turnaround times across thousands or even millions of incremental iterations. Without incremental partitioning, the overhead of repetitive full partitioning can accumulate significantly, and the benefits of partitioning cannot be fully exploited.

To overcome this limitation by adding support for incrementality, this thesis also presents two GPU-parallel incremental partitioning algorithms. Figure 0.1 illustrates the workflows of (c) iG-kway, a GPU-parallel incremental graph partitioner, and (d) iHyperG, a GPU-parallel incremental hypergraph partitioner. Both algorithms efficiently update the modified structure directly on the GPU and selectively refine only the vertices that require refinement in parallel, significantly reducing runtime by avoiding repartitioning from scratch. iG-kway features a bucket-list data structure that dynamically updates the graph directly on the GPU, along with a refinement kernel that refine many affected vertices in parallel. This design achieves significant speedup over the state-of-the-art GPU-based full graph partitioner, G-kway. On the other hand, iHyperG introduces a scalable delta-based data structure to address the high memory demands imposed by complex multi-pin relationships in hypergraphs. In addition, it employs a refinement strategy for hypergraphs that efficiently identifies cut-critical vertices within modified hyperedges and refines them in parallel, achieving substantial speedup over the existing GPU-based full hypergraph partitioner, HyperG, without compromising partitioning quality.

In the next four chapters, we provide an in-depth explanation of our GPU-accelerated partitioning algorithms: G-kway, HyperG, iG-kway, and iHyperG, respectively.

1 G-KWAY: MULTILEVEL GPU-ACCELERATED k-WAY GRAPH PARTITIONER

Graph partitioning is important for the design of many CAD algorithms. However, as the graph size continues to grow, graph partitioning becomes increasingly time-consuming. Recent research has introduced parallel graph partitioners using either multi-core CPUs or GPUs. However, the speedup of existing CPU graph partitioners is typically limited to a few cores, while the performance of GPU-based solutions is algorithmically limited by available GPU memory. To overcome these challenges, we propose G-kway, an efficient multilevel GPU-accelerated k-way graph partitioner. G-kway introduces an effective union find-based coarsening and a novel independent set-based refinement algorithm to significantly accelerate both the coarsening and uncoarsening stages. Furthermore, when kernel launch overhead becomes substantial in the refinement algorithm, G-kway employs CUDA Graph-based uncoarsening to reduce the overhead and improve performance. Experimental results have shown that G-kway outperforms both the state-of-the-art CPU-based and GPU-based parallel partitioners with an average speedup of $8.6\times$ and $3.8\times$, respectively, while achieving comparable partitioning quality. Additionally, G-kway with CUDA Graph-based uncoarsening can further accelerate graph partitioning, achieving up to $1.93\times$ speedup over the default G-kway.

1.1 Introduction

Graph partitioning is important for the design of efficient computer-aided design (CAD) algorithms because it allows an algorithm to break down a problem into smaller and manageable pieces. Among various partitioning frameworks, *multilevel partitioning* is the most popular for large-scale graphs due to its high partitioning quality and fast runtime. A typical multilevel partitioner iteratively coarsens the original graph into a smaller representation. When the graph becomes small enough, the partitioner iteratively restores the graph back to a larger one, followed by a refinement algorithm.

However, as the size of circuit graphs continues to increase, graph partitioning becomes time-consuming. To alleviate the long runtime, existing partitioners [1] have leveraged multi-core CPUs to parallelize the partitioning algorithm. Despite some runtime improvements, the speedup is typically limited to only 8–16 CPU threads [1]. On the other hand, modern GPUs offer a massive amount of parallelism and memory bandwidth that present an opportunity to accelerate graph partitioning to a new performance degree. For instance, [16] proposes a CPU-GPU-hybrid multilevel graph partitioner that dynamically performs the work on either the GPU or CPU. However, their approach requires frequent data transfers between CPU and GPU, resulting in significant runtime overhead. To address this problem, GKSG [17] performs the entire graph partitioning on a GPU. However, their performance is far from optimal due to limited parallelism. Specifically, GKSG's refinement algorithm can only move a few vertices (e.g., 8) in parallel due to limited GPU memory, as it counts on an exponential enumeration to find a valid refinement. Furthermore, GKSG's coarsening algorithm requires many sequential matching iterations, largely underutilizing the massive parallelism in GPU. As a consequence, GKSG reported only an average 1.9× speedup over a CPU-parallel partitioner [17].

To overcome these problems, we propose G-kway [18], a new GPU-accelerated k-way graph partitioner. We summarize three key contributions of G-kway below:

- G-kway introduces a union find-based coarsening algorithm that can coarsen many vertices simultaneously to substantially reduce the number of levels while keeping good partitioning quality. Specifically, at each level, our union find-based coarsening joins multiple connected vertices into the same subset and coarsens them into a single coarse vertex to construct a coarser graph, significantly reducing the graph size.

- G-kway introduces an independent set-based refinement that can refine many vertices in parallel, largely reducing the number of refinement iterations. Specifically, in each refinement iteration, our independent set-based refinement identifies thousands of independent vertices with positive gains and relocates them in parallel to improve partitioning quality.
- G-kway introduces CUDA Graph-based uncoarsening for graphs with significant kernel launch overhead, utilizing CUDA Graph and conditional nodes to reduce overhead and minimize CPU intervention, thereby enhancing performance. Specifically, CUDA Graph reduces the overhead of frequent kernel launches by encapsulating the entire computation workflow into a predefined execution graph, allowing the CPU to launch it with a single host call.

We have evaluated the performance of G-kway on industrial circuit graphs and compared our results with two state-of-the-art parallel graph partitioners, CPU-based mt-metis [1] and GPU-based GKSG [17]. On average, experimental results have shown that G-kway outperforms 32-threaded mt-metis and GKSG by 8.6× and 3.8× faster, respectively, with comparable cut sizes.

1.2 Problem Definition and Notation

Given an undirected graph, $G = (V, E)$, where V is a set of vertices, and E is a set of edges. Each element in E is of the form $e = (u, v)$ which represents the connection between u and v in V. For a vertex $v \in V$, we denote the weight of v by W_v, while for an edge $e \in E$, we denote the weight of e by W_e. For a vertex $v \in V$, its adjacent vertex set is denoted as $adj(v)$. Given k, if $P = \{p_1, p_2, \ldots, p_k\}$ is a disjoint partition of V, we call P a k-way partition. For $v \in V$, we define $P(v) = i$ if $v \in p_i$. We define the cut size as $\sum_{e=(u,v)\in E, P(u)\neq P(v)} W_e$. Cut size is widely used for evaluating the quality of a partition since it represents the interconnect complexity among partitions. The partition weight of p_i is defined as $W_{p_i} = \sum_{v \in p_i} W_v$. The goal of the graph partition problem is to find a k-way partition that satisfies the balance constraint while minimizing the cut size. The balance constraint limits the maximum weight of p_i as $W_{p_i} \leqslant (1+\epsilon)\frac{\sum_{v\in V} W_v}{k}$, where $0 < \epsilon \ll 1$ and ϵ is the imbalance ratio given by applications.

1.3 GPU Multilevel k-way Partitioner

Figure 1.1 shows the overview of G-kway that consists of three main stages: *coarsening*, *initial partition*, and *uncoarsening*.

- *Coarsening.* The goal is to coarsen the graph into a smaller representation level by level while preserving the original graph's structure. The coarsening level continues until the graph size becomes smaller than a certain threshold (typically $\frac{|V|}{20\times(\log_2(k))}$). We develop a *union find-based coarsening* that substantially reduces the number of coarsening levels while still maintaining a good representation of the original graph structure.

- *Initial partition.* The goal is to create an initial partition from the coarsest graph. We utilize single-threaded Metis [10] for the initial partition. Since the coarsest graph is much smaller than the original graph, the initial partition stage is very fast and does not benefit much from CPU/GPU parallelism.

- *Uncoarsening.* The goal is to iteratively restore the coarsened graph back to its previous graph and reduce the cut size of a coarsened graph by moving each vertex to a partition (i.e., refinement). The uncoarsening level continues until the graph size is the same as the original graph. We develop an efficient *independent set-based* refinement algorithm that reduces the cut size by moving many vertices among partitions in parallel.

Multilevel graph partitioning requires many iterative control-flow operations performed on the CPU to determine termination. Such frequent CPU-GPU data transfers can result in significant runtime overhead. To address this issue, G-kway utilizes CUDA pinned memory for control-flow data to avoid swapping out memory to disk by the operating system. In both coarsening and uncoarsening stages, we utilize modern warp-level primitives for our GPU kernels to further optimize the performance. In terms of graph storage, G-kway utilizes the commonly used compressed sparse row (CSR) data structure [17] for efficient GPU computing.

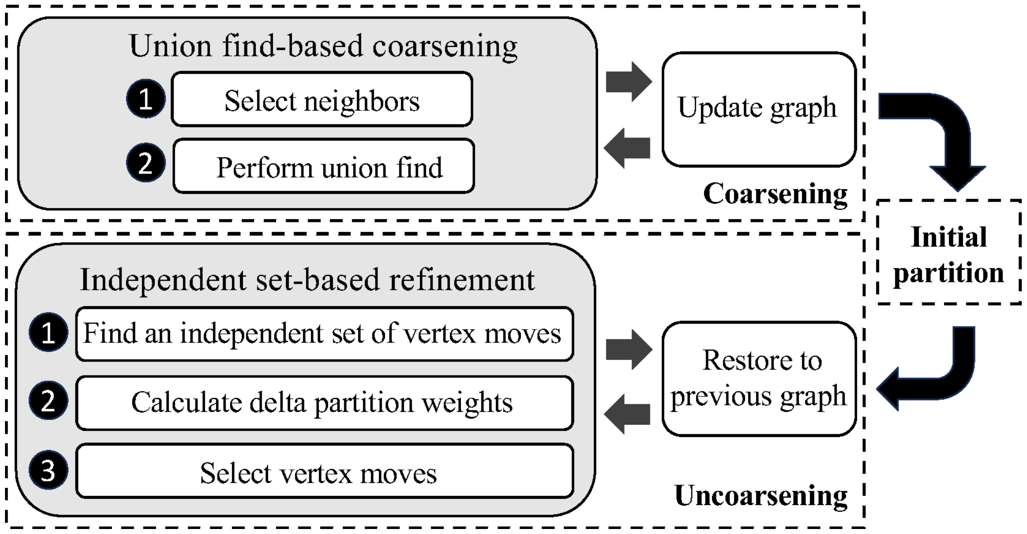

Union find-based coarsening
1
Select neighbors
2
Perform union find
Update graph
Coarsening
Initial partition
Independent set-based refinement
1
Find an independent set of vertex moves
2
Calculate delta partition weights
3
Select vertex moves
Restore to previous graph
Uncoarsening

Figure 1.1: Overview of G-kway that consists of three main stages: coarsening, initial partition, and uncoarsening.

Union Find-based Coarsening with Scoring

Most existing parallel multilevel graph partitioners such as GKSG [17] implement a parallel Heavy Edge Matching (HEM) algorithm that finds matching pairs to coarsen the original graph. Specifically, each vertex searches for a neighbor with the heaviest edge to form a matching pair and coarsen the two vertices into a coarsened vertex. However, this matching algorithm requires both vertices to choose each other. If both vertices have many neighbors connected with the same heaviest edges, they may choose different neighbors for matching, preventing the formation of matching pairs and leaving many vertices unable to match. The unmatched vertices continue to search with their remaining neighbors in the next matching iteration. Such an iterative algorithm largely underutilizes the massive parallelism in GPU. Furthermore, GKSG can only coarsen two vertices per matching pair, thus requiring many coarsening levels until the size of the coarsened graph is smaller than the threshold. Figure 1.2 shows the comparison between GKSG's coarsening algorithm and ours. As shown in (a), GKSG can only match v_1 and v_2 in the first iteration, leaving the unmatched vertices v_3 and v_4 for the next matching iteration.

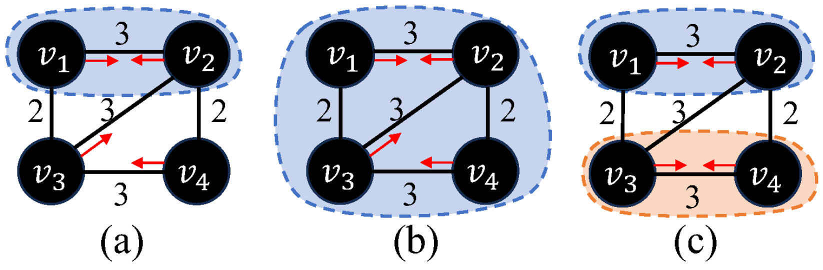


Figure 1.2: Examples of three coarsening methods for one iteration, including (a) Heavy Edge Matching (HEM) by GKSG, (b) Union find-based coarsening without scoring, and (c) Union find-based coarsening with scoring. Each vertex has a red arrow pointing to its selected neighbor. Vertices circled in the same color are coarsened into a coarsened vertex.

To address these issues, our initial solution is to merge vertices into subsets and coarsen all vertices in the same subset into a coarsened vertex. Each vertex finds a neighbor with the heaviest edge. If that neighbor belongs to another subset, we merge the vertex into the same subset. This union find-based strategy eliminates the need for iteratively searching neighbors to match, ensuring each vertex can find a neighbor to merge in only one iteration. Also, since we merge multiple vertices per subset, it requires much fewer coarsening levels than GKSG. However, this strategy can cause highly imbalanced subsets that largely impact refinement quality in the next stage since many vertices may all be merged into the same subset. As shown in 1.2 (b), v_1 and v_3 choose v_2, v_2 chooses v_1, and v_4 chooses v_3. While the solution allows each vertex to find a neighbor in one iteration, all vertices are eventually merged together.

To this end, we propose a union find-based coarsening with scoring. Each vertex calculates the score for each connected edge and selects a neighbor with the highest score to form a subset. Specifically, when a vertex u has multiple neighbors with the same heaviest edge, we prioritize the neighbor of u with the lower degree by assigning a higher score to the edge connected to this neighbor. Figure 1.2 (c) shows our union find-based coarsening with scoring. v_3 selects $v4$ instead of v_2 since v_4 has lower degree than v_2, resulting in two balanced subsets. Our coarsening algorithm consists of two steps: *select neighbors* and *perform union find*.

Select Neighbors

We first find a neighbor connected by the edge with the highest score for each vertex. Given a source vertex u, we define the score of its edge (u, v) as $s(u, v) = c \times W_{(u,v)} - degree(v)$, where $degree(v)$ is the number of neighbors of v, c is a constant no less than the maximum degree of the graph, and $W_{(u,v)}$ is the edge weight of $e = (u, v)$. Algorithm 1 shows our neighbor selection algorithm which leverages an efficient *Warp segmentation* technique [19]. We assign 32 consecutive vertices and their edges to each GPU warp. Each GPU thread then processes an edge (u, v) by finding the source vertex (line 5) and calculating the edge score (line 6). Next, threads whose assigned edges belong to the same source vertex perform parallel reduction to identify the edge with the highest score (line 8). During reduction, we employ CUDA warp-level primitives, `__shfl_up_sync`, to efficiently exchange scores among threads in the same warp. Using the warp-level primitive allows threads in the same warp to share data through registers, which is much faster than through GPU global or share memory [19]. Finally, we map each thread to a vertex, and each thread is responsible for writing a vertex's neighbor connected by the highest-score edge to the array $selected\_nbr$ in the GPU's global memory (line 10).

Perform Union Find

After selecting the highest score neighbor for each vertex, we perform `union_find` to merge vertices into a subset. We maintain an additional array, d_subset_ID, to record each vertex's subset ID, where each vertex's subset ID is initialized to its vertex ID. We assign each vertex v_i to a GPU thread; then, each thread gets its assigned vertex's selected neighbor from the previous step stored in $selected\_nbr$ (line 15), and its vertex and selected neighbor's subset IDs from d_set_ID (lines 16-17). Each thread then merges vertices by comparing its assigned vertex and the selected neighbor's subset ID and changing the larger ID to the smaller one (lines 18-21). At the end of each iteration, we employ CUDA warp-level voting primitives, `__any_sync`, to efficiently check if any thread in the warp updates the subset ID (line 23). We then repeat this process until no vertex's subset ID is updated. Finally, we coarsen vertices with the same subset ID into a coarsened vertex to derive the coarsened graph.

Algorithm 1: Union Find-based Coarsening with Scoring

```
/* select neighbors: assign 32 vertices and their edges to a GPU warp */
parallel for each thread in a warp {
    while (there are more edges to process) {
        get an edge $e_i = (u, v)$ to process
        find the assigned edge's source vertex $u$
        $s(u, v) \leftarrow c \times W_{(u,v)}$ - degree($v$)
        /* using __shfl_up_sync */
        reduce on the scores with threads have the same source vertex
    }
    write a vertex $u$'s selected neighbor to $selected\_nbr$ array
}
/* union find: assign each vertex $v_i$ to a GPU thread $T_i$ */
while (any threads is still updating) {
    parallel for each thread in a warp {
        $nbr \leftarrow selected\_nbr[v_i]$
        $v_i\_subset\_ID \leftarrow d\_subset\_ID[v_i]$
        $nbr\_subset\_ID \leftarrow d\_subset\_ID[nbr]$
        if ($v_i\_subset\_ID > nbr\_subset\_ID$) then
            atomicMin(&$d\_subset\_ID[v_i]$, $nbr\_subset\_ID$)
        else if ($v_i\_subset\_ID < nbr\_subset\_ID$) then
            atomicMin(&$d\_subset\_ID[nbr]$, $v_i\_subset\_ID$)
        /* using __any_sync */
        check if any thread in a warp updates subset_IDs
    }
}
```

Independent Set-based Refinement

The goal of the refinement algorithm is to reduce the cut size by moving a vertex to a partition, seeking the maximum gain in cut size reduction. We define the gain of a vertex u for a partition p_i as $gain(u, p_i) = ed(u, p_i) - id(u)$, where $u \notin p_i$. $id(u)$ represents the internal degree of u, which is the sum of the weights of each edge (u, v) such that u and v are in the same partition. $ed(u, p_i)$ represents the external degree of u to partition p_i, which is the sum of the weights of each edge (u, v) such that

v is in partition p_i. In refinement, we only consider moving a vertex at the partition boundary (i.e., one of its neighbors is located in a different partition). Moving vertices not at the partition boundary cannot have positive gain, as $ed(u, p_i)$ is always zero.

To move multiple vertices in parallel while ensuring that the move results in the largest gain, GKSG's refinement algorithm enumerates all possible moves [17]. Each move represents a combination of vertices, where each vertex either moves to its destination partition or not. For example, to move eight vertices in parallel, GKSG will launch a GPU kernel with $2^8 \times 32$ threads to calculate 2^8 possible moves, where each move is verified by a GPU warp of 32 threads. This *exponential* enumeration algorithm limits the number of vertices that can be moved in parallel due to the limited GPU memory.

To overcome this problem, we propose an independent set-based refinement algorithm that can move many vertices in parallel. Our algorithm does not exponentially enumerate all possible moves, thus enabling much more parallelism without being constrained by GPU memory limitations. Algorithm 2 shows our refinement algorithm, which contains three steps: *find an independent set of vertex moves*, *calculate delta partition weights*, and *select vertex moves*. We iteratively perform our refinement algorithm until no vertex with positive gain can be moved.

Find an Independent Set of Vertex Moves

Moving multiple vertices in parallel is challenging. Even though each vertex has a positive gain, the overall cut size after all moves can remain or even increase due to interconnections among vertices. Furthermore, moving connected (i.e., adjacent) vertices in parallel requires expensive synchronization to keep updating gains. To address these issues, we find an independent set of vertices to move in parallel. We define each *vertex move* as $m_u^{src,dst}$, a struct that consists of a vertex ID (u), its source partition

ID (`src`), its destination partition ID (`dst`), and the gain. We then use a move buffer to store vertex moves.

Algorithm 2 presents our independent set-based refinement algorithm. To find an independent set of vertex moves, we distribute each vertex in the graph to a GPU thread, where each GPU thread determines whether its vertex is at the partition boundary. If the vertex is at the boundary, the GPU thread finds a legal destination partition for that vertex (line 7).

We say a vertex has a legal destination partition if there exists one destination partition such that moving the vertex to that partition has a positive gain without violating the balance constraint. If a vertex has a legal destination partition, the GPU thread checks if any of its neighbors also have a legal destination partition (line 9). If no such neighbor exists, the GPU thread creates a vertex move for the vertex and inserts it into the move buffer (lines 10-12). Otherwise, we compare that vertex with its neighbors' IDs. We then only create a vertex move for the vertex with the smallest ID and insert it into the move buffer (lines 13-16). This organization ensures that no adjacent vertices are inserted into the move buffer.

After finding an independent set of vertex moves, we need to select a subset of them such that applying those vertex moves still satisfies the balance constraint. However, finding the best subset still encounters the exponential enumeration problem (i.e., to select or not to select per vertex move). To address this challenge, we design a sequence-based strategy that first sorts each vertex move by gain to form a sequence and selects the longest sub-sequence of vertex moves that satisfies the balance constraint. While this strategy may not be the absolute best subset, selecting vertex moves from the largest gain ensures we prioritize the vertex moves that make a substantial contribution to overall cut size improvement. In the following sections, we present how to find that sub-sequence of vertex moves.

Algorithm 2: Independent Set-based Refinement

```
 1: while (true) {
 2:     /* find an independent set of vertex moves */
 3:     /* assign each vertex $v_i$ to a GPU thread $T_i$ */
 4:     parallel for each thread {
 5:         if $v_i$ is not at a partition boundary then
 6:             return
 7:         $dst \leftarrow$ find a legal destination partition with the largest gain
 8:         if ($dst$ exists) then
 9:             $nbors \leftarrow nbr$ in $adj(v_i)$ has a legal destination partition
10:             if ($nbors$ is empty) then
11:                 create a vertex move for $v_i$
12:                 insert the vertex move to the move buffer
13:             else
14:                 if ($v_i$.ID < each $nbr$.ID in $nbors$) then
15:                     create a vertex move for $v_i$
16:                     insert the vertex move to the move buffer
17:     }
18:     if (the move buffer is empty) then
19:         return
20:     calculate delta partition weights      /* Section 3.2.2 */
21:     select vertex moves                    /* Section 3.2.3 */
22: }
23: return
```

Calculate delta partition weights

In this step, we sort each vertex move by gain in descending order and calculate the delta partition weight of each vertex move to check the balance constraint. We define the delta partition weight of a vertex move $m_u^{src,dst}$ for a partition p_i as follows:

$$\delta_i(m_u^{src,dst}) = \begin{cases} W_u, & i = dst \\ -W_u, & i = src \\ 0, & otherwise \end{cases}$$

We maintain a k-segment array, del_p_wgt, where each segment initially stores the delta partition weight of each vertex move for a partition. The segment size is the minimum of the total number of vertex moves and 1024. Since most modern GPUs have 1024 threads per GPU block, calculating more than 1024 vertex moves needs multiple blocks for each segment, which requires expensive synchronization across multiple blocks.

Figure 1.3 shows an example of our algorithm for six vertex moves with $k = 2$. Each element in del_p_wgt records the delta partition weight of each of the six vertex moves, where the first six elements (i.e., segment 0) and the last six elements (i.e., segment 1) are for partitions p_0 and p_1, respectively. We then perform a parallel scan on del_p_wgt to accumulate delta partition weights for each partition. Specifically, after applying the parallel scan, the j^{th} element in segment s stores the accumulated delta partition weight from the first to the j^{th} vertex moves for partition s (i.e., a sub-sequence from the first to the j^{th} vertex moves). This accumulation allows us to quickly access each partition's accumulated delta partition weight if we apply all vertex moves in a sub-sequence of vertex moves. We then use these accumulated results to find the longest sub-sequence of vertex moves in the next step.

Algorithm 3 presents the calculation of delta partition weights. We first sort vertex moves in the move buffer by gain in descending order in parallel using a parallel sorting algorithm (line 1). We assign each vertex move, $m_u^{src,dst}$, to a GPU thread, T_i based on its gid. Each GPU thread first gets the index of a vertex move's source (src_p_idx) and destination partition (des_p_idx) in $delta\_p\_wgt$ (lines 6-7). Each GPU thread then writes the corresponding delta partition weights to del_p_wgt (lines 8-9).

Finally, we apply our parallel scan kernel on each segment to obtain the accumulated delta partition weights per partition (lines 13-18). We launch our parallel scan kernel with the number of GPU blocks equal to k (i.e., number of partitions), where each GPU block conducts a parallel scan simultaneously for its assigned segment (line 15). To further improve

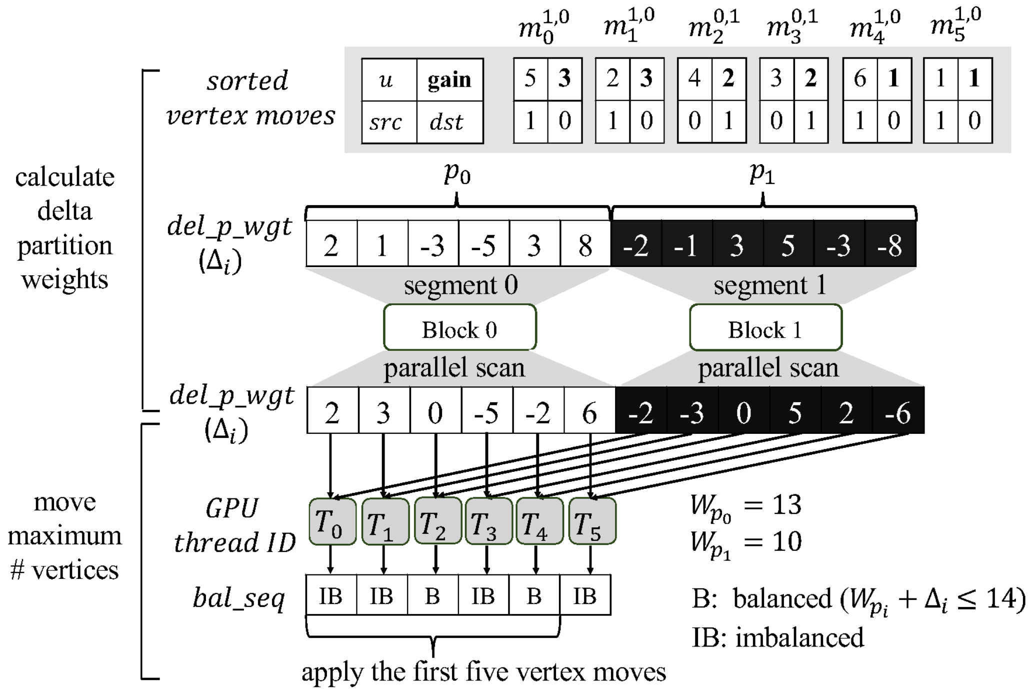


Figure 1.3: Illustration of the process to construct del_p_wgt and bal_seq with k = 2 under six vertex moves. Assuming current partition weights are 13 and 10 for p_0 and p_1, respectively, with a balance constraint of 14.

performance, we utilize a CUDA warp-level primitive, `__shfl_up_sync`, for our parallel scan kernel.

Select Vertex Moves

In this step, we select the longest sub-sequence of vertex moves while ensuring that applying those vertex moves satisfies the balance constraint. This selection is based on our accumulated delta partition weights.

As shown in Figure 1.3, we maintain a bal_seq array to record the balanced condition for a sub-sequence of vertex moves. The value stored at index j in bal_seq indicates whether applying the sub-sequence of vertex moves from the first to the jth results in a balanced partition. We then select the longest sub-sequence of vertex moves by finding the largest

Algorithm 3: Calculate Delta Partition Weights

```
1:  **parallel sort** the move buffer in descending order by gain
2:  $seg\_size \leftarrow \min(\#vertex\_moves, 1024)$
3:  $gid \leftarrow$ thread's global ID
4:  /*assign a vertex move $m_u^{dst}$ to a GPU thread $T_i$ based on its $gid$*/
5:  **parallel for each thread {**
6:      $src\_p\_idx \leftarrow m_u^{src,dst}.src \times seg\_size + gid$
7:      $dst\_p\_idx \leftarrow m_u^{src,dst}.dst \times seg\_size + gid$
8:      $del\_p\_wgt[src\_p\_idx] \leftarrow -W_u$
9:      $del\_p\_wgt[dst\_p\_idx] \leftarrow W_u$
10:     **return**
11: }
12: /*assign segment $seg_i$ of del_p_wgt to a GPU block $b_i$*/
13: **parallel for each block {**
14:     $seg_i\_start \leftarrow b_i.ID \times seg\_size$
15:     $seg_i\_end \leftarrow seg_i\_start + seg\_size$
16:     **parallel scan** on $seg_i$                /* __shfl_up_sync */
17:     **return**
18: }
```

index j such that $bal\_seq[j] = B$ (balanced). Finally, we apply all vertex moves in the longest sub-sequence of vertex moves.

In the example shown in Figure 1.3, each GPU thread checks if a sub-sequence of vertex moves results in a balanced partition and writes the result to the bal_seq array. Specifically, the first thread (T_0) checks the balanced result for the sub-sequence of vertex moves of the first vertex move, the second thread (T_1) checks for the sub-sequence of vertex moves from the first to the second vertex moves, and so on. Each thread fetches the accumulated delta partition weight for each partition from each segment in del_p_wgt, and checks whether every partition's current weight plus its accumulated delta partition weight satisfies the balance constraint. For example, assuming the balance constraint is 14, T_0 fetches $del\_p\_wgt[0]$ and $del\_p\_wgt[6]$ for p_0 and p_1, and checks if both $W_{p0} + \Delta_0 \leqslant 14$ and $W_{p1} + \Delta_6 \leqslant 14$. If one of the partitions does not satisfy the balance constraint, the thread writes 'IB' (imbalanced); otherwise, 'B' (balanced) to its corresponding index in bal_seq.

After each thread finalizes bal_seq, we can observe that applying only the first vertex move results in an imbalanced partition ($bal\_seq[0] = \mathrm{IB}$). However, applying the first five vertex moves helps to restore the partition result back to balance ($bal\_seq[4] = \mathrm{B}$). In the example shown in Figure 1.3, the longest sub-sequence is from the first to the fifth vertex moves. Since the sequence of vertex moves is sorted by gain in descending order, we can prioritize those vertex moves that make a substantial contribution to the overall improvement in cut size. Finally, we apply all vertex moves in the longest sub-sequence of vertex moves in parallel.

Accelerating the Uncoarsening Stage Using CUDA Graph

In this section, we discuss how CUDA Graphs can be beneficial for the uncoarsening stage. In the uncoarsening stage, the refinement algorithm iteratively applies vertex moves to improve the partitioning result and terminates when no more vertex moves can be applied. For large benchmarks, the number of refinement iterations can be substantial (e.g., 2,000), leading to significant kernel launch overhead and degraded performance. To mitigate this overhead, we encapsulate all GPU operations within a CUDA Graph. The CPU (i.e. host) can then launch the CUDA Graph to perform each uncoarsening level with a single call. This approach significantly reduces kernel launch overhead and minimizes CPU intervention, accelerating the uncoarsening stage. Figure 1.4 illustrates the time spent on kernel execution with and without a CUDA Graph. Without CUDA Graphs, the host needs to launch each kernel individually, and each kernel launch incurs overhead. When the number of iterations is large, the accumulated overhead can become a bottleneck. In contrast, using CUDA Graphs significantly reduces kernel launch overhead by allowing the host to launch all GPU kernels with a single host call. In this section, we first introduce the CUDA Graph execution model and discuss the recently added feature, the CUDA Graph conditional node. Finally, we describe the implementation details of our CUDA Graph-based uncoarsening.

CUDA Graph-based Execution Model

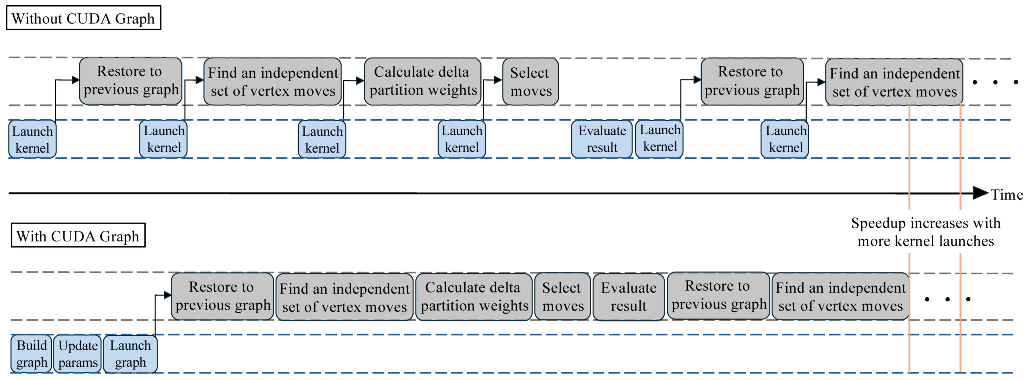

Without CUDA Graph
Restore to previous graph
Find an independent set of vertex moves
Calculate delta partition weights
Select moves
Restore to previous graph
Find an independent set of vertex moves
Launch kernel
Launch kernel
Launch kernel
Launch kernel
Evaluate result
Launch kernel
Launch kernel
Time
Speedup increases with more kernel launches
With CUDA Graph
Restore to previous graph
Find an independent set of vertex moves
Calculate delta partition weights
Select moves
Evaluate result
Restore to previous graph
Find an independent set of vertex moves
Build graph
Update params
Launch graph

Figure 1.4: The time spent on kernel execution with and without CUDA Graphs. Each gray box represents a GPU task, while the blue box indicates a CPU operation. The top graph shows the traditional approach without using CUDA Graphs, where each kernel launch incurs significant overhead. The bottom graph shows the approach using CUDA Graphs, which reduces launch overhead by encapsulating kernel executions into a single graph, leading to an overall speedup in execution time.

In CUDA, each time the host launches a GPU kernel, a small overhead incurs. When a large number of kernels are launched, this overhead can accumulate and become significant, degrading overall performance. To address this issue, the CUDA Graph execution model was introduced to enable more efficient execution of GPU kernels [20]. CUDA Graph allows users to encapsulate all GPU kernels within a graph, so the host can execute them with a single call (i.e. graph launch), significantly reducing kernel launch overhead. Additionally, encapsulating GPU kernels into a CUDA Graph allows the CUDA driver to optimize the entire graph as a whole, further enhancing performance [21, 22, 23, 24]. Fig 1.5 presents the CUDA Graph execution model, which consists of graph definition, executable instantiation, graph launch (run), and graph update. Users first define a CUDA Graph by creating nodes and edges to describe GPU tasks and their dependencies. Users then instantiate an executable graph from a defined graph and offload that executable graph to a GPU using a single host call. Between successive launches, users can update the execution parameters of a GPU task or a node in the graph with a small cost.

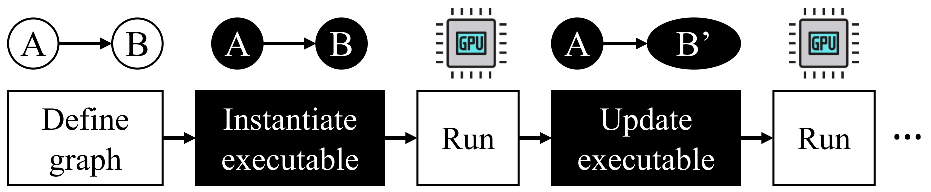


Figure 1.5: Execution model of CUDA Graph consists of four major steps, graph definition, executable graph instantiation, graph launch (run), and graph parameter updates.

CUDA Graph with Conditional Node

While CUDA Graphs offer significant performance benefits, constructing one requires users to know the graph's topology at compile time. If a graph involves dynamic control flow, where certain node launches depend on a control variable, users must isolate those nodes into a separate CUDA Graph. The host then evaluates the control variable to determine whether to launch the separate graph. This approach limits CUDA's ability to optimize all GPU kernels as a single CUDA Graph, ties up CPU resources, and introduces additional overhead from setting up multiple graphs.

To address this issue, CUDA recently introduced conditional nodes in CUDA Graph, enabling conditional or repeated launch of graph nodes without returning to the host. Each conditional node contains a body graph, whose execution depends on a control variable. At runtime, the conditional node evaluates the control variable and determines whether to launch its body graph, allowing dynamic control flow directly on the GPU. There are two types of conditional nodes: an *if* node and a *while* node. The body graph of an *if* node will be executed once if the condition is met, while the body graph of a *while* node will be executed repeatedly as long as the condition is true. Using conditional nodes helps minimize CPU intervention and allows more complex workflows to be represented within a CUDA Graph, eliminating the overhead of creating multiple graphs and reducing the number of graph launches required. Figure 1.6 illustrates the workflow of a sequence of GPU tasks involving a *while* dynamic control flow, without and with a conditional node. In Figure 1.6 (a), the absence of a conditional node requires separating the GPU tasks into two CUDA Graphs, with GPU tasks involving dynamic control flow isolated into a second graph. After the host launches the first CUDA Graph, it must iteratively evaluate the condition and launch the second CUDA Graph, introducing significant graph launch overhead. In contrast, in Figure 1.6 (b) with a conditional node, all GPU tasks can be defined within a CUDA Graph. The host can then launch the defined graph with a single call, greatly reducing the overhead associated with iterative graph launches.

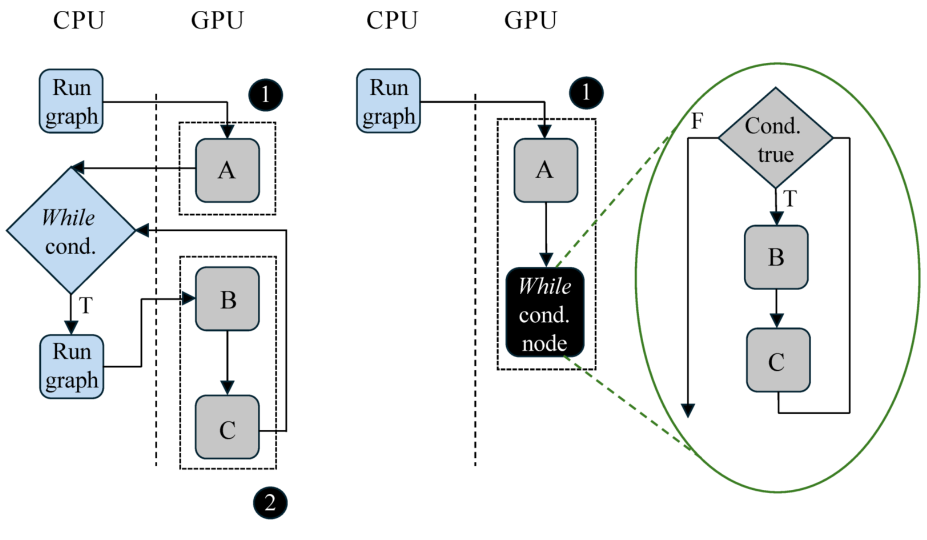


Figure 1.6: An example of the workflow of a sequence of GPU tasks involving a *while* dynamic control flow. In Figure (a), the workflow is shown without a CUDA conditional node, and in Figure (b), the workflow is shown with a CUDA conditional node. Each gray box represents a graph node that contains a GPU task, while a blue box represents a CPU operation. The black box represents a CUDA conditional node. Graph nodes belonging to the same graph are grouped using a black dashed line.

CUDA Graph for Uncoarsening Stage

In G-kway, both the uncoarsening and coarsening stages involve a sequence of kernel launches. However, the number of kernel launches in the coarsening stage is typically small. Therefore, using CUDA Graph does not accelerate the coarsening stage, as its setup overhead outweighs its performance gains. In contrast, during the uncoarsening stage, the refinement algorithm iteratively moves vertices to refine the partitioning result, often requiring a large number of iterations (e.g., 2,000). Each iteration involves multiple GPU kernel launches, accumulating to significant kernel launch overhead. To mitigate this, we encapsulate all GPU kernels used in the uncoarsening algorithm into a CUDA Graph. This approach allows the host to launch all GPU operations with a single call, improving

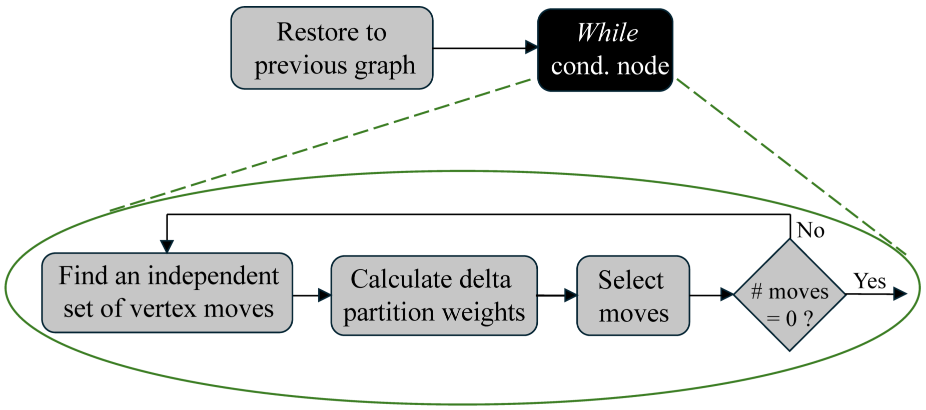

Restore to previous graph
While cond. node
Find an independent set of vertex moves
Calculate delta partition weights
Select moves
# moves = 0 ?
No
Yes

Figure 1.7: The CUDA Graph encapsulates all GPU kernels used for uncoarsening. Within the graph, a CUDA conditional node iteratively evaluates the number of vertex moves to manage dynamic control flow on the GPU.

performance by minimizing host intervention and reducing the number of host calls. Additionally, the CUDA driver can optimize the CUDA Graph, further enhancing performance.

To create the uncoarsening CUDA Graph, we use a *while* conditional node to manage the iterative control flow in the refinement algorithm. This conditional node then evaluates the number of vertex moves applied in the previous iteration and repeats the refinement process as long as the number of vertex moves remains greater than zero. Figure 1.7 illustrates the topology of our uncoarsening CUDA Graph. In the graph, the first node is responsible for executing the GPU kernel *restores to previous graph*. Then, the *while* conditional node is executed. The body graph of this conditional node contains three GPU tasks: *find an independent set of vertex moves*, *calculate delta partition weights*, and *select vertex moves*.

Algorithm 4 outlines the process of creating an uncoarsening CUDA Graph. We first create the graph using `cudaGraphCreate` (line 2). Next, we define a graph node to execute *restore to previous graph* kernel (line 3) and specify this node's parameters by setting the grid and block dimensions and kernel arguments in a `cudaKernelNodeParams` structure

Algorithm 4: Create Uncoarsening CUDA Graph

```
 1: Initialize a CUDA Graph: $graph$
 2: cudaGraphCreate(&$graph$, 0)
        /* Create a node for restore to previous graph kernel and add it to the CUDA Graph */
 3: Initialize a CUDA Graph node: $restore\_graph\_node$
 4: $kernel\_params$ ← { 0 }
 5: set_kernel_params($kernel\_params$)
 6: cudaGraphAddKernelNode(&$restore\_graph\_node$, $graph$, NULL, 0, $kernel\_params$)
        /* Create a CUDA condition node and add it to the graph */
 7: Initialize a CUDA Graph conditional handle: $handle$
 8: cudaGraphConditionalHandleCreate(&$handle$, $graph$, 1, cudaGraphCondAssignDefault)
 9: $while\_params$ ← { cudaGraphNodeTypeConditional }
10: $while\_params$.conditional.handle ← $handle$
11: $while\_params$.conditional.type ← cudaGraphCondTypeWhile
12: $while\_params$.conditional.size ← 1
13: Initialize a CUDA Graph conditional node: $while\_cond\_node$
14: cudaGraphAddKernelNode(&$while\_cond\_node$, $graph$, &$restore\_graph\_node$, 1, $while\_params$)
15: $while\_body\_graph$ ← $while\_params$.conditional.phGraph_out[0]
        /* Populate the condition node's body graph */
16: Initialize a CUDA Graph node: $find\_independent\_node$
17: set_kernel_params($kernel\_params$)
18: cudaGraphAddKernelNode(&$find\_independent\_node$, $while\_body\_graph$, NULL, 0, $kernel\_params$)
        /* ... Add other nodes using similar method */
19: return $graph$
```

named `kernel_params` (line 5). We then add this node to the CUDA graph, `graph` (line 6). To manage the control flow in the refinement algorithm, we first create a CUDA conditional node by defining a CUDA Graph conditional handle [20] and attaching it to `graph` (lines 7-8). Next, we define a `cudaGraphNodeParams` structure named *while_params* to store the necessary information for this conditional node. In *while_params*, we specify that this conditional node is a while node and provide its size and handle (lines 9-12). We then add this conditional node to `graph` (lines 13-14) and set its dependency so that it runs after the graph node, *restore_graph_node*. When the conditional node is created, its body graph is also created. We retrieve the body graph from *while_params* (line 15) and populate it by creating nodes for each kernel used in the refinement algorithm and adding them to the body graph using a similar method (lines 16-19).

Algorithm 5: Uncoarsening stage with CUDA Graph

```
1: graph ← creat_uncoarsening_cuda_graph()
2: initialize graph_exec and stream
3: cudaGraphInstantiate (&graph_exec, graph, NULL, NULL, 0)
4: for each uncoarsening level {
5:     update_graph_paramaters()      ▷ update the graph nodes' parameters
6:     cudaGraphLaunch(graph_exec, stream)  ▷ launch the executable graph
7: }
8: cudaGraphDestroy(graph)
```

Once the CUDA Graph for the uncoarsening algorithm is created, the host can launch it with a single call at each uncoarsening level, significantly reducing kernel launch overhead and minimizing CPU intervention. Algorithm 5 outlines our CUDA Graph-based uncoarsening. Before starting the uncoarsening stage, the CPU host first calls the function `create_uncoarsening_cuda_graph` (as illustrated in Algorithm 4) to obtain the uncoarsening CUDA Graph (line 1). Next, the host instantiates the graph into an executable using `cudaGraphInstantiate` (line 3).

Finally, the host begins the uncoarsening stage by iteratively uncoarsening the graph level by level. At each uncoarsening level, the host first updates the CUDA Graph's node parameters (line 5) and then uses a single call to launch the executable graph with the updated parameters (line 6).

1.4 Computational Complexity of Partitioning Algorithms

In the previous section, we discussed G-kway's innovative union find-based coarsening and independent set-based refinement for accelerating the coarsening and uncoarsening stages. Building on this discussion, we now evaluate the efficiency of these algorithms by comparing the time complexity of G-kway's coarsening and uncoarsening stages with two state-of-the-art parallel partitioners: mt-metis [1] (CPU-based) and GKSG [17] (GPU-based). Table 1.1 summarizes the time complexity of the coarsening and uncoarsening stages for mt-metis, GKSG, and G-kway. In the following sections, we first analyze and compare the time complexity of the coarsening stage among the three partitioners, followed by a discussion of the uncoarsening stage.

Table 1.1: Time complexity analysis of the coarsening and uncoarsening stages for three parallel graph partitioners: mt-metis (CPU-based), GKSG (GPU-based), and G-kway. The number of vertices is denoted as $|V|$, and B is the buffer size. The parameters T_{mt}, l_{mt}, and np_{mt} denote the number of available threads, levels, and refinement passes for mt-metis, respectively. Similarly, T_{GKSG}, l_{GKSG}, it_{GKSG} and T_{gk}, l_{gk}, it_{gk} represent the corresponding values for GKSG and G-kway, with it indicating refinement iterations.

Partitioner	Coarsening Stage	Uncoarsening Stage
mt-metis	$O(l_{mt} \times \frac{\lvert V\rvert}{T_{mt}})$	$O(l_{mt} \times np_{mt} \times k \times \frac{\lvert V\rvert}{T_{mt}})$
GKSG	$O(l_{GKSG} \times \frac{\lvert V\rvert}{T_{GKSG}})$	$O(l_{GKSG} \times it_{GKSG} \times (k \times \frac{\lvert V\rvert}{T_{GKSG}} + \frac{B \times \log(B)}{T_{GKSG}}))$
G-kway	$O(l_{gk} \times \frac{\lvert V\rvert}{T_{gk}})$	$O(l_{gk} \times it_{gk} \times (k \times \frac{\lvert V\rvert}{T_{gk}} + \frac{B \times \log(B)}{T_{gk}}))$

Coarsening Stage Comparison

During the coarsening stage, the coarsening algorithm reduces the graph size level by level until it falls below the coarsening threshold, γ. In the coarsening algorithm of mt-metis, vertices are distributed among CPU threads, with each thread responsible for finding a matched neighbor for its assigned vertex. Thus, the time complexity of this coarsening algorithm is $\frac{|V|}{T_{mt}}$, where $|V|$ is the number of vertices in the graph and T_{mt} is the number of CPU threads. However, since the coarsening algorithm is applied at each level, the time complexity of mt-metis's coarsening stage is

$$O(l_{mt} \times \frac{|V|}{T_{mt}})$$

, where l_{mt} represents the number of levels required for mt-metis to coarsen the graph below γ.

For GKSG, the time complexity of the coarsening stage is similar to that of mt-metis, as it assigns 32 vertices to a GPU warp (32 threads). Thus, the time complexity of GKSG's coarsening algorithm is given by:

$$O(l_{GKSG} \times \frac{|V|}{T_{GKSG}})$$

, where l_{GKSG} denotes the number of levels required by GKSG, and T_{GKSG} is the total number of available GPU threads. However, since GKSG utilizes significantly more GPU threads compared to the CPU threads in mt-metis, $\frac{|V|}{T_{GKSG}}$ is much smaller than $\frac{|V|}{T_{mt}}$, leading to a reduction in the coarsening time.

For G-kway, its coarsening algorithm has a similar time complexity to that of GKSG, as both assign a GPU warp to cooperatively process 32 vertices. However, unlike GKSG and mt-metis, which only allow two vertices to be coarsened into a single coarsened vertex, G-kway's union find-based algorithm groups multiple vertices into the same subset and coarsens them together in parallel. This approach significantly reduces the number of required levels, l_{gk} (i.e. $l_{gk} \ll l_{GKSG} \approx l_{mt}$) by substantially decreasing the graph size at each level, thereby improving the efficiency of the coarsening stage.

Uncoarsening Stage Comparison

During the uncoarsening stage, the refinement algorithm is applied at each level to improve the partitioning result. In mt-metis, the refinement algorithm consists of multiple passes, where in each pass vertices are distributed among CPU threads. Each thread determines the destination partition of its assigned vertices by calculating the gain for each boundary partition, resulting in up to k gain calculations. Thus, the time complexity of mt-metis's refinement algorithm is given by $np_{mt} \times k \times \frac{|V|}{T_{mt}}$, where k denotes the number of partitions, np_{mt} represents the number of passes, and T_{mt} is the number of available CPU threads. Additionally, since the refinement algorithm is executed at each level, the total time complexity of mt-metis's uncoarsening stage is

$$O(l_{mt} \times np_{mt} \times k \times \frac{|V|}{T_{mt}}).$$

In GKSG, the refinement algorithm assigns each GPU thread to a vertex, determines its destination partition, and inserts vertices with positive gains into the buffer. The time complexity of this process is $k \times \frac{|V|}{T_{GKSG}}$. However, to identify the vertices with the highest gains, GKSG requires an additional step to sort the buffer. This sorting step has a time complexity of $\frac{B \times \log_(B)}{T_{GKSG}}$, where B represents the buffer size. At each uncoarsening level, GKSG iteratively executes its refinement algorithm until no further vertex movements improve the partitioning. Therefore, the total time complexity is

$$O(l_{GKSG} \times it_{GKSG} \times (k \times \frac{|V|}{T_{GKSG}} + \frac{B \times \log(B)}{T_{GKSG}}))$$

, where $\mathtt{it}_{\mathtt{GKSG}}$ is the number of refinement iterations. Since $\mathsf{T}_{\mathsf{GKSG}}$ is much larger than $\mathsf{T}_{\mathtt{mt}}$, GKSG's refinement algorithm processes vertices in parallel more efficiently than mt-metis, resulting in a faster runtime despite the additional sorting cost.

For G-kway, the refinement algorithm has a similar time complexity to that of GKSG, as it also uses a buffer to store and identify the vertices with the highest gains. However, unlike GKSG, which can move only 8–16 vertices per refinement iteration due to memory limitations imposed by its exponential enumeration algorithm, G-kway can move thousands of vertices simultaneously in parallel. As a result, the number of refinement iterations in GKSG is significantly smaller than in GKSG (i.e., $\mathtt{it}_{\mathtt{gk}} \ll \mathtt{it}_{\mathtt{GKSG}}$), leading to a substantial speedup in the uncoarsening stage.

1.5 Experimental Evaluation

We evaluated the performance of G-kway on six industrial circuit graphs generated by [25], where regular graphs are used to represent timing graphs. Additionally, we tested G-kway's performance on four large non-circuit graphs (ldoor, NLR, delaunay, asia.osm) from the DIMACS Graph Partitioning Challenge to demonstrate our applicability beyond CAD algorithms. We implemented G-kway using C++17 and CUDA 12.0 and compiled it with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz and 256 GB RAM. Our GPU is A6000 with 48 GB global memory.

Baselines

We consider mt-metis v0.7.2 [1] and GKSG [17] as baseline partitioners. Mt-metis is a state-of-the-art CPU-parallel graph partitioner that renovates the sequential Metis algorithm [10] to a parallel target using OpenMP.

GKSG is a state-of-the-art GPU-accelerated graph partitioner. Since GKSG is not open-source, we implemented its algorithm on our GPU except for the initial partitioning. Because the coarsest graph is typically very small, we do not observe any advantage in using GPU. In all experiments, we set the imbalance ratio to 3% and the coarsening threshold to $\frac{|V|}{20\times(\log_2(k))}$. These settings are the same as the default values of mt-metis [1] and GKSG [17] that can produce the best results. All data is an average of ten runs.

Overall Performance Comparison

Table 1.2 compares the overall runtime and cut size results among G-kway, GKSG, and mt-metis at $k = 2$. We ran mt-metis using 32 threads to achieve the best performance on our machine. In terms of runtime, G-kway outperforms GKSG and mt-metis across all graphs, with an average speedup of 3.8× and 8.6×, respectively. The largest speedups we observe are 9.1× over GKSG in asia.osm and 14.3× over mt-metis in wb_dma. The significant improvement on runtime demonstrates the promise of our union find-based coarsening and independent set-based refinement algorithms. For the smallest graph, ldoor, G-kway still achieves 6.5× and 1.6× over GKSG and mt-metis. We attribute this significant speedup to our efficient coarsening algorithm that efficiently coarsens many vertices per subset, thus largely reducing the number of coarsening levels. Regarding cut size, G-kway outperforms mt-metis and GKSG on nearly all graphs. For instance, on vga_lcd, our cut size is 3.6× better than mt-metis. We attribute this improvement to our coarsening algorithm, which results in better-coarsened graphs. Similar improvements can be found when comparing G-kway with GKSG.

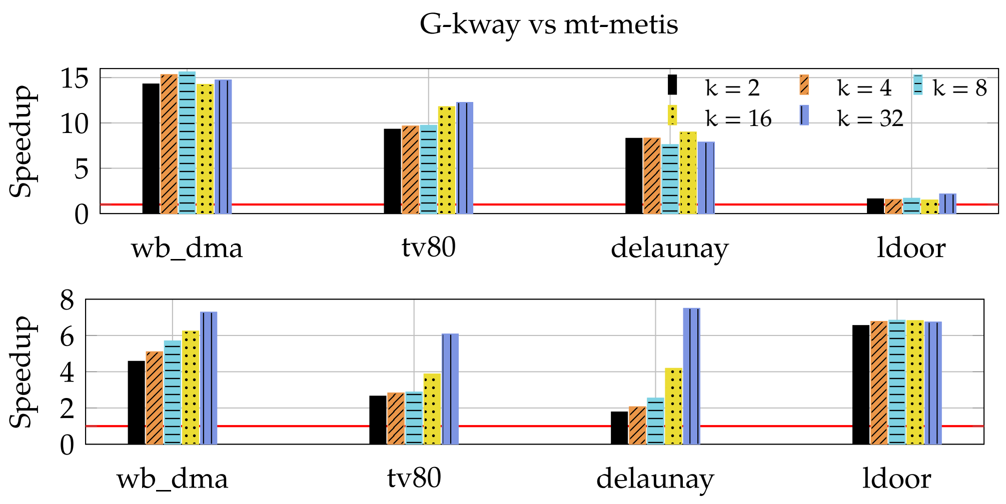


Figure 1.8: The speedup of G-kway over mt-metis (top) and GKSG (bottom) at different k.

Runtime Analysis

Figure 1.8 shows the speedup of G-kway over mt-metis (32 threads) and GKSG with different k on two circuit graphs (wb_dma, tv80) and two non-circuit graphs (delaunay, ldoor). Regardless of k, G-kway is always faster than mt-metis and GKSG. Compared to mt-metis, G-kway achieves over 6× and 10× for more than 80% and 40% of the partitioning problem instances, respectively. For large graphs, such as wb_dma, our speedups are remarkable. The proposed GPU-accelerated coarsening and refinement algorithms bring significant performance benefits to parallel graph partitioning. Similar speedup values can also be observed in the comparison with GKSG. For instance, G-kway is 7× faster than GKSG on the wb_dma with $k = 32$.

Cut Size Analysis

Figure 1.9 shows the cut size improvement ratio of G-kway over mt-metis and GKSG at $k = \{2, 4, 8, 16, 32\}$. In general, G-kway can produce parti-

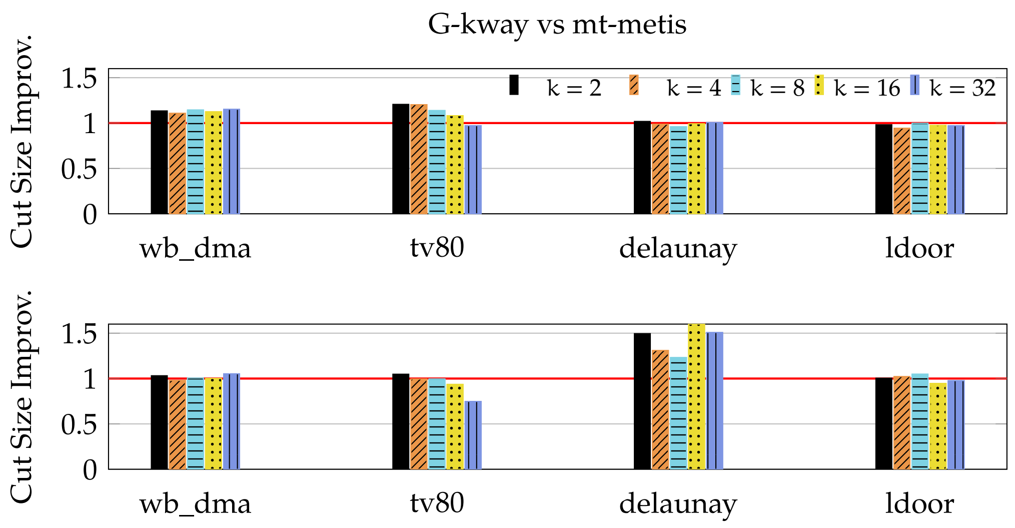


Figure 1.9: The cut size improvement ratio of G-kway over mt-metis (top) and GKSG (bottom) at different k.

tions with comparable quality to mt-metis and GKSG. Compared to GKSG, G-kway finds partitions with significantly less cut size for delaunay. We attribute this to our refinement algorithm. GKSG can only move a few vertices (e.g., eight) at one refinement iteration due to the memory limitation of its exponential enumeration algorithm. On the other hand, our refinement algorithm identifies a sequence of vertices through independent set finding and identifies the longest sub-sequence that satisfies the balance constraint. This approach allows G-kway to discover more valid moves in one iteration that can lead to a better cut size. However, moving too many vertices simultaneously can sometimes trap us in a local minima that produces a worse cut size than GKSG, such as tv80 at $k = 32$. Compared to other graphs, tv80 has longer path connectivity among vertices which can benefit from more fine-grained refinement as GKSG.

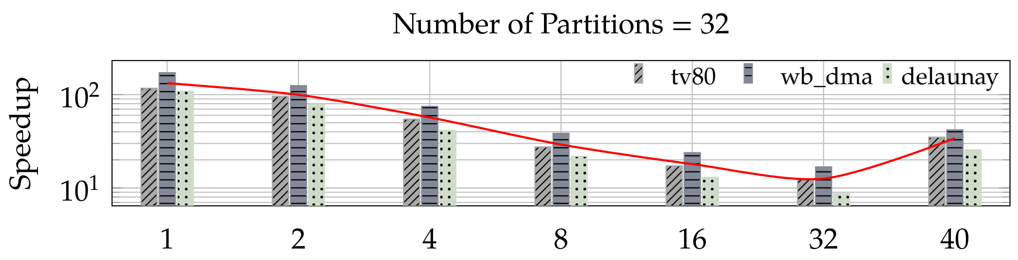


Figure 1.10: The speedup of G-kway over mt-metis at various numbers of threads for tv80, wb_dma, and delaunay at $k = 32$. The red line indicates the average speedup trend of the three graphs.

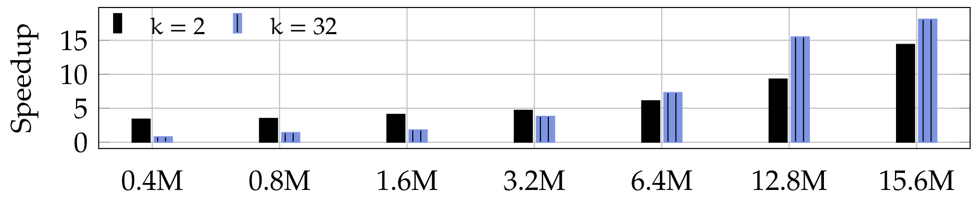


Figure 1.11: The speedup of G-kway over mt-metis at varying graph sizes modified from usb at $k = 2$ and $k = 32$.

Absolute Efficiency over mt-metis

Figure 1.10 shows the speedup of G-kway over mt-metis using the different number of CPU threads at $k = 32$. Regardless of the thread count, G-kway is always faster. For example, G-kway is 172× and 16× faster than mt-metis using one and 32 threads. We observe that the performance of mt-metis begins to saturate at about 32 threads and becomes worse beyond. For instance, using 40 threads is 20% slower than using 32 threads in mt-metis. We believe this problem comes from both the internal threading overhead of mt-metis and the limitation of CPU parallelism on throughput optimization when processing large graph data. Figure 1.11 illustrates the speedup of G-kway over mt-metis (32 threads) on partitioning varying circuit sizes at two extreme k, 2 and 32. We randomly remove the vertices and edges of usb to generate different graph sizes from 400K to 15.6M. Below 400K, we do not see much runtime difference between mt-metis and G-kway. However, as the graph size becomes larger than 1M vertices, we can see the absolute efficiency of GPU acceleration over CPU-based mt-metis. The speedup of G-kway continues to enlarge as we increase the graph size.

Analysis of Coarsening with and without Scoring

Table 1.3 compares the cut size between G-kway with scoring (G-kway) and G-kway without scoring (G-kway^{-s}) to study the effectiveness of the proposed scoring-based coarsening. Compared to G-kway^{-s}, G-kway achieves better cut size at all k. G-kway^{-s} fails to find a solution that meets the balance constraint for the highly connected ldoor at $k = 8$ and $k = 32$. Without scoring, G-kway^{-s} groups many vertices into the same subset, resulting in a highly imbalanced coarsened graph. Such an imbalance greatly impacts the partition results at later initial partition and refinement stages.

Table 1.3: Cut size comparison in terms of reduction (↓) between G-kway with scoring (G-kway) and G-kway without scoring (G-kway^{-s}) for ldoor and delaunay at k= {2, 8, 32}.

	ldoor		delaunay	
k	G-kway^{-s}	G-kway	G-kway^{-s}	G-kway
2	44,064	25,578 (↓**72%**)	10,381	8,463 (↓**23%**)
8	✗	101,639	38,799	33,673 (↓**15%**)
32	✗	290,225	96,274	80,609 (↓**19%**)

Analysis of Coarsening Threshold on Partitioning Performance

Table 1.4 compares the cut size and runtime achieved by G-kway when partitioning two circuit graphs (pci_bridge and tv80) and two non-circuit graphs (NLR and delaunay) at two extreme k values, 2 and 32, with three coarsening thresholds (γ): k, $160 \times k$, and $\frac{|V|}{20 \times \log_2(k)}$. All data is an average of ten runs.

When γ is set to k, G-kway fails to produce a balanced partition for the NLR and delaunay graphs at $k = 2$ and for all graphs at $k = 32$. Setting γ to a very small value (e.g., k) requires many levels to sufficiently reduce the graph size. At later levels, vertex connections become denser as vertices are coarsened. This increase in density causes G-kway's union find-based algorithm to coarsen many vertices together, leading to imbalances in coarsened vertex sizes that make the later initial partitioning and uncoarsening stages struggle to find a balanced partition. On the other hand, setting γ to $160 \times k$ and $\frac{|V|}{20 \times \log_2(k)}$ produces a similar cut size. However, setting $\gamma = \frac{|V|}{20 \times \log_2(k)}$ increases the overall partitioning time, as γ depends on the input graph size. For large graphs, γ can become significantly large, making the initial partitioning computationally expensive. To achieve the best performance, we set γ to $160 \times k$ for the rest of our experiments.

Performance Enhancement Due to CUDA Graph

To evaluate CUDA Graph's effectiveness in accelerating the uncoarsening stage, we selected six benchmarks with different numbers of refinement iterations. Table 1.5 lists the size and maximum number of refinement iterations for the six benchmarks used to evaluate G-kway with CUDA Graph-based uncoarsening (G-kway^g). Additionally, to demonstrate the importance of accelerating the uncoarsening stage, we analyzed the time distribution across the three partitioning stages—coarsening, initial partitioning, and uncoarsening—using the three largest circuit graphs. Figure 1.12 shows the time distribution for partitioning vga_lcd, wb_dma, and aes_core at $k = 2$ and $k = 32$. We observe that for large circuit graphs, uncoarsening can account for more than 80% of the total partitioning time due to the large number of refinement iterations. The iterative kernel launches for refining vertices introduce substantial overhead, making CUDA Graph particularly effective in reducing this overhead and improving performance.

Figure 1.13 shows the speedup of G-kway^g over the default G-kway during the uncoarsening stage for $k = \{2, 8, 16, 32, 64\}$. On average, G-kway^g achieves a 1.27× speedup over G-kway, regardless of the value of k. The largest speedup we observed is 1.93× for NLR at $k = 64$. However, for the benchmark asia.osm, which has a small number of refinement iterations, G-kway^g has a longer runtime than G-kway. This is because the accumulated kernel launch overhead in asia.osm is relatively small, and the overhead of setting up the CUDA Graph outweighs its benefits. Conversely, once the number of refinement iterations exceeds 20, the accumulated kernel launch overhead becomes significant, and using CUDA Graph consistently speeds up the uncoarsening stage.

Table 1.5: A list of the number of vertices, edges, and the maximum number of refinement iterations for six selected benchmarks used to analyze CUDA Graph acceleration. The maximum number of refinement iterations is the highest observed across ten runs for each value of $k = \{2, 8, 16, 32, 64\}$

Benchmark	# Vertices	# Edges	Maximum # Refinement Iterations
NLR	4,163,763	12,487,976	32
AS365	3,799,275	11,368,076	35
wb_dma	131,240	275,936	1,899
vga_lcd	795,612	1,302,327	6,698
aes_core	200,253	322,340	2,052
asia.osm	1,1950,757	12,711,603	15

.

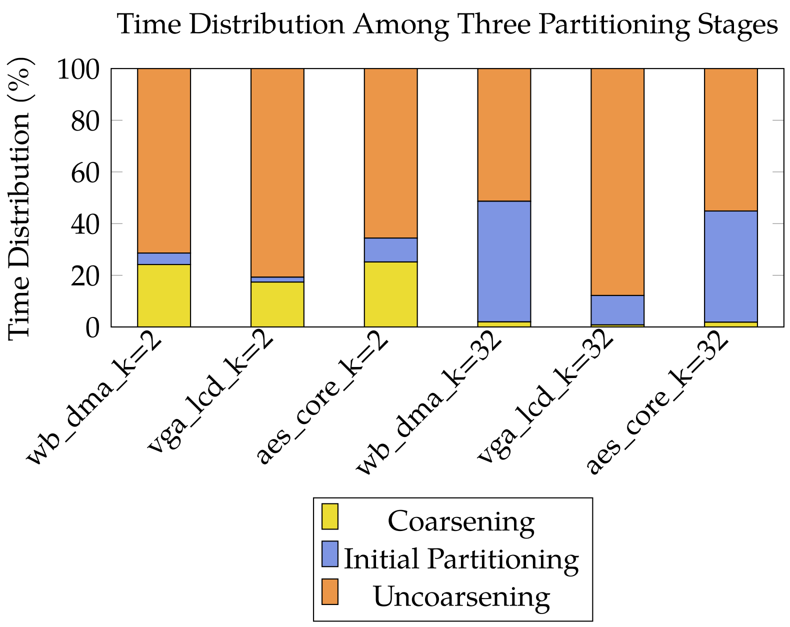


Figure 1.12: Time distribution among the coarsening, initial partitioning, and uncoarsening stages for the three largest circuit graphs—wb_dma, vga_lcd, and aes_core—at k = 2 and k = 32.

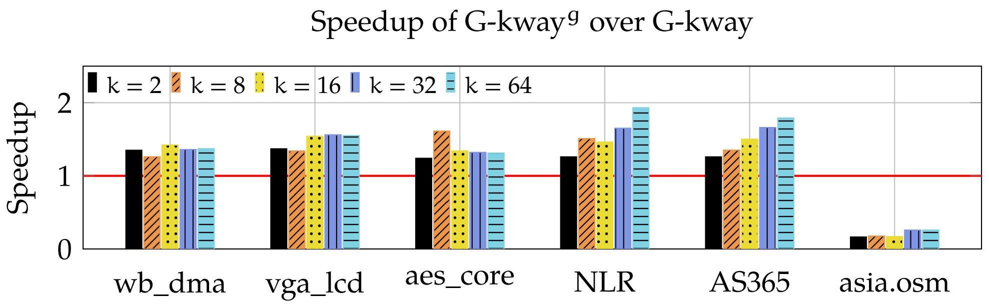


Figure 1.13: The speedup of G-kway with CUDA Graph-based uncoarsening, G-kway^g, over the default G-kway during the uncoarsening stage at k = {2, 8, 16, 32, 64}.

CUDA Graph Time Breakdown

Figure 1.14 illustrates the breakdown of time spent on three graph operations—graph creation & update, graph launch, and graph instantiation—for the benchmark with the largest number of refinmenet iterations, vga_lcd. Among the three operations, graph launch is the most time-consuming operation. Furthermore, because this operation is called multiple times, the cumulative time spent on graph launches accounts for the majority (87%) of the total execution time. Graph instantiation is the second most expensive operation; even though G-kway^{g} only instantiates the graph once, this operation contributes 12% to the total graph execution time. On the other hand, the cost associated with creating and updating the CUDA Graph is minimal, accounting for just 1% of the overall execution time.

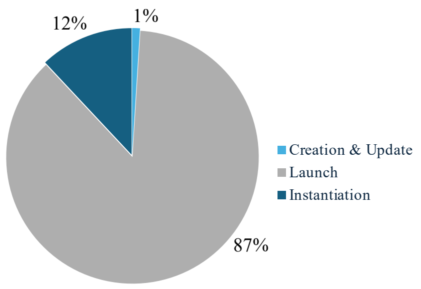


Figure 1.14: The breakdown of total graph execution time spent on each operation, including creation & update, launch, and instantiation, for vga_lcd in G-kway^{g}.

Analysis of CUDA Graph with and without CUDA Graph Conditional Nodes

Table 1.6 compares the number of host calls for graph creation, instantiation, and launch between G-kway^{g}, which employs a CUDA Graph with a conditional node, and G-kway$^{g,-\mathrm{cond}}$, which employs a CUDA Graph without a conditional node. All data is an average of ten runs. With a conditional node, G-kway^{g} enables the GPU to manage the iterative control flow of the refinement algorithm, encapsulating all GPU kernels within a single CUDA Graph. As a result, only one graph creation and instantiation call is needed, and each uncoarsening level requires just a single host call to launch the uncoarsening graph. This approach minimizes the number of host calls, leading to a more efficient execution of the uncoarsening stage. In contrast, without a conditional node, G-kway$^{g,-cond}$ requires the host to manage the iterative control flow by isolating the GPU kernels that involve control flow into a separate CUDA Graph. Consequently, for all benchmarks, G-kway$^{g,-cond}$ needs twice as many graph creation and instantiation calls as G-kway^{g}, introducing additional overhead in setting up the CUDA Graphs. Furthermore, for each iteration, the host must iteratively evaluate the control flow condition and launch the graph involved in the control flow. As a result, G-kway$^{g,-cond}$ invokes graph launches numerous times, leading to substantial overhead due to frequent host intervention and graph launches.

Table 1.7: Cut size and runtime (in seconds) comparisons with varying imbalance ratio (ϵ) of 0.03 and 0.3, when partitioning graphs at two extreme k values: 2 and 32.

Benchmark	$k = 2$				$k = 32$			
	$\epsilon = 0.03$		$\epsilon = 0.3$		$\epsilon = 0.03$		$\epsilon = 0.3$	
	Cut Size	Time	Cut Size	Time	Cut Size	Time	Cut Size	Time
pci_bridge	4,188	0.14	4,001	0.14	61,612	0.32	8,618	0.17
tv80	2,422	0.15	2,302	0.15	8,141	0.17	5,353	0.16
NLR	4,519	0.14	4,477	0.14	43,592	0.15	43,101	0.15
delaunay	9,664	0.21	9,194	0.21	83,053	0.23	80,673	0.23

Impact of Imbalance Ratio on Partitioning Performance

Table 1.7 compares the cut size and runtime of G-kway when partitioning two circuit graphs (pci_bridge and tv80) and two non-circuit graphs (NLR and Delaunay) at two extreme values of k (2 and 32), with two imbalance ratios ϵ: 0.03 and 0.3. All data is an average of ten runs. Regardless of the k value, a higher imbalance ratio enables G-kway to consistently achieve a better cut size by allowing more vertices to move across partitions, thereby improving partition quality. However, while a larger imbalance ratio allows G-kway to move more vertices to enhance partition quality, it does not increase partitioning time. We attribute this efficiency to G-kway's independent set-based refinement, which can simultaneously move thousands of vertices.

1.6 Conclusion

In this chapter, we have introduced G-kway, an efficient GPU-accelerated multilevel k-way graph partitioner. G-kway features a union find-based coarsening algorithm that significantly reduces the number of levels and an independent set-based refinement algorithm that can move many vertices in parallel. For graphs with significant kernel launch overhead, G-kway leverages CUDA Graph-based coarsening to reduce the overhead and further enhance performance. Experimental results have shown that G-kway outperforms the state-of-the-art CPU-based and GPU-based parallel partitioners by 8.6× and 3.8× faster while achieving comparable partitioning quality. Additionally, G-kway with CUDA Graph-based uncoarsening can further accelerate graph partitioning, achieving up to 1.93× speedup over the default G-kway.

In this work, Wan Luan Lee was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and over- sight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Table 1.2: Overall comparison of runtime (seconds) and cut size among GKSG, mt-metis (32 threads), and G-kway at $k = 2$. The last four columns represent the speedup and cut size improvement of G-kway over GKSG and mt-metis, respectively.

Benchmark			GKSG		mt-metis		G-kway		Speedup vs		Cut size improv. vs	
Name	# Vertices	# Edges	Time (s)	Cut size	Time (s)	Cut size	Time (s)	Cut size	GKSG	mt-metis	GKSG	mt-metis
pci_bridge	12,394,539	15,809,551	0.89	5,114	3.21	4,773	**0.27**	**4,293**	3.5×	12.5×	1.2	1.1
vga_lcd	1,392,3210	24,904,499	1.33	5,661	3.80	17,237	**0.46**	**4,737**	2.9×	8.4×	1.2	3.6
wb_dma	19,686,000	20,236,297	1.21	7,131	3.81	7,844	**0.32**	**6,915**	4.6×	14.3×	1.0	1.1
usb	25,215,939	31,630,268	2.03	7,933	6.97	10,283	**0.58**	**6,709**	3.5×	12.0×	1.2	1.5
tv80	13,102,222	17,759,671	0.70	2,575	2.46	3,094	**0.26**	**2,457**	2.6×	9.3×	1.0	1.3
mem_ctrl	6,422,461	8,455,835	0.62	7,368	1.35	7,126	**0.26**	**6,976**	2.4×	5.3×	1.1	1.0
ldoor	952,203	2,278,5136	0.84	25,676	0.21	**25,088**	**0.13**	25,578	6.5×	1.6×	1.0	1.0
NLR	4,163,763	12,487,976	0.16	4,432	0.34	**4,262**	**0.15**	4,705	1.1×	2.3×	0.9	0.9
delaunay	16,777,216	50,331,601	0.48	12,650	2.23	8,614	**0.27**	**8,463**	1.8×	8.3×	1.5	1.0
asia.osm	11,950,757	12,711,603	0.96	9	1.30	8	**0.11**	**7**	9.1×	12.4×	1.3	1.1
Average									3.8×	8.6×	1.1	1.4

Table 1.4: Cut size and runtime (in seconds) comparisons with three coarsening thresholds (γ): k, $160 \times$ k, and $\frac{|V|}{20\times \log_2(k)}$, for partitioning graphs at two extreme k values: 2 and 32. If G-kway fails to find a balanced partition, the result is denoted as ✗.

Benchmark	k = 2						k = 32					
	γ = k		γ = 160×k		$\gamma = \frac{|V|}{20\times \log_2(k)}$		γ = k		γ = 160×k		$\gamma = \frac{|V|}{20\times \log_2(k)}$	
	Cut Size	Time	Cut Size	Time	Cut Size	Time	Cut Size	Time	Cut Size	Time	Cut Size	Time
pci_bridge	4,271	0.15	4,188	0.14	4,293	0.27	✗	✗	60,075	0.32	60,094	0.40
tv80	2,416	0.15	2,422	0.15	2,457	0.26	✗	✗	7,596	0.17	7,869	0.25
NLR	✗	✗	4,519	0.14	4,262	0.15	✗	✗	43,592	0.15	45,030	0.16
delaunay	✗	✗	83,482	0.21	8,463	0.27	✗	✗	83,053	0.23	81,320	0.25

Table 1.6: The maximum number of host calls for graph creation, instantiation, and launch for G-kway^{g} and G-kway$^{g,-\mathtt{cond}}$ was observed across ten runs for each value of k = {2, 8, 16, 32, 64}. Both G-kway^{g} and G-kway$^{g,-\mathtt{cond}}$ implemented CUDA Graph-based uncoarsening. However, G-kway^{g} uses a CUDA Graph conditional node to manage control flow on the GPU, while G-kway$^{g,-\mathtt{cond}}$ omits the conditional node, relying on the host for control flow management.

Benchmark	G-kway^{g}			G-kway$^{g,-\mathtt{cond}}$		
	Graph Creation	Graph Init	Graph Launch	Graph Creation	Graph Init	Graph Launch
wb_dma	1	1	3	2	2	1,902
vga_lcd	1	1	4	2	2	6,702
aes_core	1	1	3	2	2	2,055
AS365	1	1	6	2	2	41
NLR	1	1	6	2	2	38
asia.osm	1	1	6	2	2	21

2 HYPERG: MULTILEVEL GPU-ACCELERATED K-WAY HYPERGRAPH PARTITIONER

Hypergraph partitioning plays a critical role in computer-aided design (CAD) because it allows us to break down a large circuit into several manageable pieces that facilitate efficient CAD algorithm designs. However, as circuit designs continue to grow in size, hypergraph partitioning becomes increasingly time-consuming. Recent research has introduced parallel hypergraph partitioners using multi-core CPUs to reduce the long runtime. However, the speedup of existing CPU parallel hypergraph partitioners is typically limited to a few cores. To overcome these challenges, we propose HyperG, a GPU-accelerated multilevel k-way hypergraph partitioning algorithm. HyperG introduces an innovative balanced group coarsening and a sequence-based refinement algorithm to accelerate both the coarsening and uncoarsening stages. Experimental results show that HyperG outperforms both the state-of-the-art sequential and CPU-based parallel partitioners with an average speedup of 133× and 4.1× while achieving comparable partitioning quality.

2.1 Introduction

Hypergraph partitioning plays a critical role in various stages of circuit design, including placement, routing, timing analysis, and logic simulation. For instance, hypergraph partitioning helps optimize component placement on a chip by dividing the circuit into smaller, more manageable blocks while minimizing interconnections among them [4]. Given that hypergraph partitioning is NP-hard [2, 4], many heuristics have been developed [9, 7, 8]. Among these heuristics, *multilevel partitioning* stands out as the most popular for large-scale circuit graphs due to its high-quality partitioning results and fast runtime [14, 2, 26, 27, 5]. A typical multilevel hypergraph partitioner iteratively coarsens the original hypergraph into a smaller representation. When the hypergraph becomes small enough, the partitioner uses a fast algorithm to generate an initial balanced partition. Finally, the partitioner iteratively restores the graph to the previous level, followed by a refinement algorithm to improve the partition solution. Among all stages, coarsening and refinement stages are the most time-consuming and account for 90% of the total runtime [17].

As the circuit size continues to grow, multilevel hypergraph partitioning becomes increasingly time-consuming. For example, the sequential hypergraph partitioner hMetis can take four minutes to partition a five-million-gate circuit [14]. Since hypergraph partitioning can be performed multiple times during a CAD algorithm (e.g., incremental timing [11], RTL simulation [12]), the cumulative partitioning time can extend to several hours. To reduce the long runtime, existing partitioners [2, 26] have utilized multi-core CPUs to parallelize the partitioning. Among many parallel graph partitioners, Mt-KaHyPar [2] is the state-of-the-art CPU-based parallel hypergraph partitioner designed to parallelize the sequential k-way Fiduccia-Mattheyses algorithm [13]. Despite some runtime improvements, the speedup typically plateaus at 8–16 CPU threads [2]. On the other hand, modern GPUs provide a massive amount of parallelism and higher memory bandwidth than CPUs, offering a new opportunity to accelerate hypergraph partitioning.

However, existing CPU parallel hypergraph partitioning algorithms cannot be directly applied to a GPU. The distinct performance characteristics between CPU and GPU require very different designs of data layouts to make the most of GPU computing. Furthermore, applying CPU-parallel algorithms to a GPU can result in underutilized GPU threads, load imbalance, and expensive synchronization overhead. For example, Mt-KaHyPar's coarsening algorithm requires frequent synchronization, which is costly on the GPU and becomes a large performance bottleneck.

Recent research, such as G-kway [18] and GKSG [17], has investigated the use of a GPU to accelerate *non-hypergraph* partitioning (i.e., exactly two vertices per edge). To apply G-kway or GKSG to hypergraph partitioning, one potential solution is to transform a hypergraph into a non-hypergraph [28]. However, this transformation often results in poor partitioning quality, as the transformed non-hypergraph fails to accurately represent the original hypergraph [14, 29]. Another possible solution is to extend G-kway's non-hypergraph partitioning algorithm to hypergraphs. However, due to the inherent differences between nonhypergraphs and hypergraphs, this approach can result in extremely low parallelism. For example, G-kway identifies an independent set of vertices to refine in parallel. Since a hyperedge can easily link to many vertices, the size of an independent set in a hypergraph is typically small. This situation becomes even more challenging with modern circuits, as a net typically connects to numerous pins. *Given these challenges and the importance of a fast hypergraph partitioner, there is a need for a new GPU-accelerated hypergraph partitoning algorithm.*

Consequently, we present *HyperG*, a GPU-accelerated hypergraph partitioning algorithm. While the previous work [30] has attempted to accelerate the uncoarsening stage using GPUs, to the best of our knowledge, HyperG is among the earliest attempts to parallelize both coarsening and uncoarsening stages on a GPU. The three key contributions of HyperG are summarized below:

- We introduce a balanced group coarsening algorithm that groups many vertices into balanced subgroups to ensure high partitioning quality at later initial partitioning stage.
- We introduce a sequence-based refinement algorithm that simultaneously identifies and moves the best subsequence of vertices to largely improve the partition solution.

- We develop our GPU kernel using modern CUDA warp-level primitives to achieve fine-grained synchronization and efficient communication during both the coarsening and refinement stages.

We evaluated the performance of HyperG on industrial circuit graphs and compared our results with two state-of-the-art hypergraph partitioners, sequential partitioner hMetis [14] and CPU-based parallel partitioner Mt-KaHyPar [2]. On average, experimental results show that HyperG outperforms hMetis and 16-threaded Mt-KaHyPar by $133\times$ and $4.1\times$ faster, respectively, with comparable cut sizes.

2.2 Problem Definition and Notation

Given a hypergraph $H = (V, E)$, where V is a set of vertices and E is a set of hyperedges, each element e in E is a subset of the vertex set V representing multi-vertex relationships. The vertices that belong to a hyperedge are referred to as that hyperedge's pins. We donate the size of e as $|e|$, which is equal to the number of pins belonging to e. For a vertex v, we denote the weight of $v \in V$ by W_v, while for a hyperedge $e \in E$, we denote the weight of e by W_e. Vertices u and v are neighbors if there exists a hyperedge $e \in E$ such that $u \in e$ and $v \in e$.

Given an integer k, the goal of the hypergraph partitioning problem is to partition V into k disjoint subsets $p_1, p_2, \dots, p_k$ of approximately equal sizes, while minimizing the cut size. The cut size is a commonly used metric to measure the interconnection among partitions and is defined as the sum of the weights of all cut hyperedges. A cut hyperedge is a hyperedge that contains pins belonging to more than one partition. For a vertex u, we define $P(u) = i$ if $v \in p_i$. The weight of the partition p_i is defined as $W_{p_i} = \sum_{v \in p_i} W_v$. To ensure that each partition has roughly equal sizes, a balance constraint is imposed, limiting the maximum size of each partition p_i as $W_{p_i} \leqslant (1+\epsilon)\frac{\sum_{v \in V} W_v}{k}$, where $0 < \epsilon \ll 1$ and ϵ is the imbalance ratio given by applications.

2.3 GPU Multilevel Hypergraph Partitioner

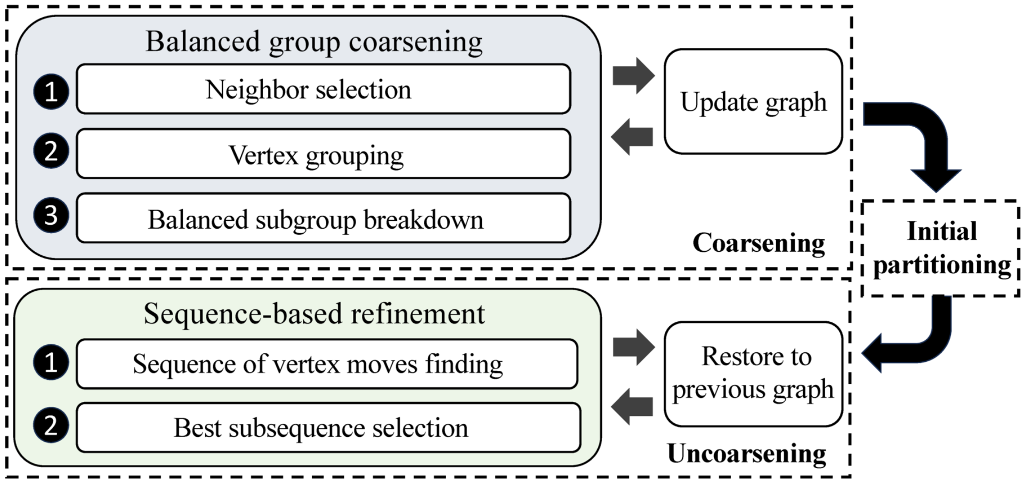

Balanced group coarsening
1 Neighbor selection
2 Vertex grouping
3 Balanced subgroup breakdown
Update graph
Coarsening
Initial partitioning
Sequence-based refinement
1 Sequence of vertex moves finding
2 Best subsequence selection
Restore to previous graph
Uncoarsening

Figure 2.1: Overview of HyperG that consists of three main stages: coarsening, initial partitioning, and uncoarsening.

Following the multilevel heuristic, HyperG consists of three main stages: *coarsening*, *initial partitioning*, and *uncoarsening*. Figure 2.1 shows an overview of HyperG.

- *Coarsening.* The goal is to coarsen the hypergraph into a smaller representation level by level while preserving the original hypergraph's structure. The coarsening process continues until the hypergraph has fewer than $160 \times k$ vertices or until less than 95% of the vertices can be coarsened in the previous level. We develop a *balanced group coarsening* algorithm that can coarsen many vertices simultaneously while ensuring that the coarsened vertices are balanced in size. This balance in vertex sizes is crucial for achieving a balanced initial partition in the initial partitioning stage.

- *Initial partitioning.* The goal is to create an initial partition from the coarsest hypergraph. We utilize the CPU-based parallel hypergraph partitioner Mt-KaHyPar [2] for the initial partitioning. Since the coarsest hypergraph is much smaller than the original hypergraph, the initial partitioning stage is very fast and does not benefit much from GPU parallelism.

- *Uncoarsening.* The goal is to iteratively restore the coarsened hypergraph back to the previous level and refine the partitioning result by moving vertices among partitions (i.e., refinement). The uncoarsening process continues until the hypergraph size is the same as the original hypergraph. We develop an efficient *sequence-based refinement* algorithm that finds the best subsequence of vertices to move in parallel, significantly improving the cut size while reducing the refinement time.

In terms of graph storage, HyperG maintains two arrays in the commonly used compressed sparse row (CSR) data structure to store vertices and their connected edges, as well as hyperedges and their connected vertices, for efficient GPU computing.

Balanced Group Coarsening

State-of-the-art CPU-parallel coarsening methods adopt rating function coarsening [2, 31, 5]. Each vertex initially starts in its own group. Each thread then visits a vertex, finds the neighbor with the highest score, and joins the vertex to that neighbor's group. Finally, all vertices in the same group will coarsen into a coarsened vertex. To prevent a group from becoming too large and causing an imbalanced initial partition at a later stage, Mt-KaHyPar imposes a maximum size on each group and prevents vertices from joining a group that exceeds the maximum size. Consequently, frequent checks (i.e., synchronization) are required to ensure that vertices do not join oversized groups. Such synchronization can introduce significant overhead for GPU, as each GPU thread involves frequent waiting for another GPU thread to update the correct group size, which reduces the parallel efficiency of GPU.

Furthermore, Mt-KaHyPar assigns each vertex an atomic variable to track each vertex's status. To ensure correct group sizes and group assignments, Mt-KaHyPar uses compare-and-swap (CAS) instructions to prevent any thread from joining a group that is currently being updated by another thread. If a thread decides to assign a vertex to a neighbor's group that is currently being joined by another thread, it enters a busy-waiting loop until the other thread finishes updating its status. Such a busy-waiting loop mechanism can significantly hamper GPU performance due to two reasons: First, busy-waiting loops reduce the number of active GPU threads performing useful computations. This can significantly hurt GPU performance since GPU relies on massive parallelism to achieve high performance. Furthermore, GPU executes threads in groups called warps. If some threads in a warp are busy-waiting while others are not, it can lead to some threads in a warp are stalled, further degrading performance.

To overcome these challenges, we propose balanced group coarsening. Each vertex selects the neighbor with the highest score to group. Each GPU thread then groups the vertices together simultaneously. Our algorithm does not limit the group size while joining vertices. Instead, We break down all groups into subgroups of similar sizes in parallel after all the vertices have joined groups. Furthermore, each GPU thread updates a vertex's group using a single *atomicMax* operation, without requiring threads to wait. Our coarsening algorithm consists of three steps: *Neighbor Selection* , *Vertex Grouping* , and *Balanced Subgroup Breakdown*.

Neighbor Selection

We develop an efficient GPU kernel using a highly optimized CUDA warp-level primitive, *__reduce_max_sync*, to find the neighbor with the highest score. Inspired by [2], we define the rating function for each vertex u and its neighbor v as follows:

$$\mathrm{score}(u, v) = C \times \sum_{e \in E : v \in e \cap u \in e} \frac{W_e}{|e|}.$$

To find the neighbor with the highest score, we assign each vertex to a GPU warp (i.e., a group of 32 consecutive threads), with each thread in the warp fetching one of the vertex's neighbors and calculating its score. We then employ *__reduce_max_sync* to perform a reduction operation across all threads in a warp to find the neighbor with the highest score. Since the built-in *__reduce_max_sync* only supports integer types, we multiply the rating function by a large constant C (i.e., 1,000). This strategy converts a float to an integer by scaling it up to a significant value, preserving the relative magnitude and precision of the original floating-point number.

Vertex Grouping

After selecting the neighbor with the highest score for each vertex, we perform the vertex grouping algorithm to join vertices into groups in parallel and coarsen all vertices in the same group together. Our vertex grouping algorithm is inspired by [32], where each vertex's group ID is iteratively updated to a larger value until all vertices with connected selected neighbors share the same group ID. However, after grouping, some groups are significantly larger than others, making it difficult to achieve a balanced partition at the later initial partitioning stage. One solution is to randomly divide each large group into smaller subgroups. However, this approach can lead to poor partition quality because vertices that are far apart may end up in the same subgroup. Coarsening these distant vertices together can distort the original graph structure. Figure 2.2 (a) shows an example of breaking down a large group into subgroups of size two randomly. This method places two nonadjacent vertices, v_1 and v_2, into the same subgroup, which distorts the original graph structure by coarsening them together.

To address these issues, we create a group combination for each vertex, ensuring that vertices closer to each other have closer group combinations. We then divide the vertices with closer group combinations into the same subgroups. In our algorithm, each vertex's group combination is an eight-byte data type, where the first four bytes store the vertex's group ID, and the last four bytes indicate the iteration during which this vertex joins the group. This setup allows us to update the group combination with a single *atomicMax* operation. Specifically, vertices closer to the group leader (i.e., the vertex with the largest vertex ID in the group) will join the group leader's group faster and adopt the smaller group combinations, while more distant vertices will join the group later and have larger group combinations. Figure 2.2 (b) shows an example of breaking down a large group into subgroups of size two by group combinations. This method places adjacent vertices into the same subgroups, which better preserves the original graph structure than method (a).

Algorithm 6 presents our vertex grouping algorithm. We use an array `group_comb` to store each vertex's group combination. Initially, each vertex is in its own group, and we initialize its corresponding value in `group_comb` such that the first four bytes equal this vertex's vertex ID and the last four bytes are set to 0. We utilize a boolean variable *joining* to track if there are any vertices still joining other groups. While *joining* is true, we assign each vertex to a GPU thread. Each thread is responsible for updating the group combinations of its assigned vertex and its selected neighbor by first comparing their group IDs and finding the larger one (line 10 or 14).

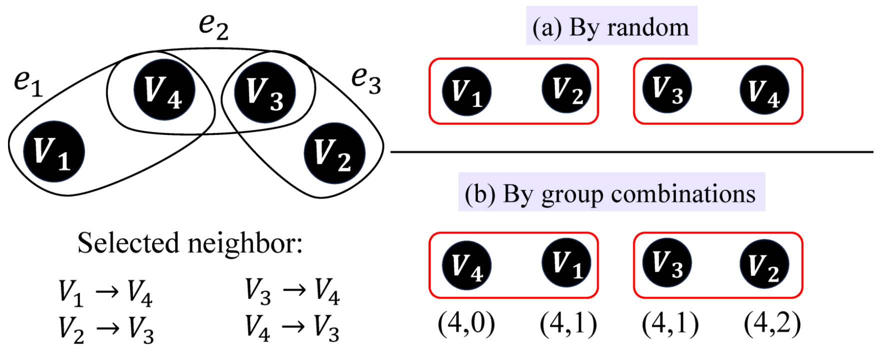


Figure 2.2: Examples of two methods to break down a large group into subgroups of size two. Vertices in the same box are placed into the same subgroup. $V_i \to V_j$ indicates V_i selects to join V_j's group. In (b), (n_1, n_2) represents a vertex's group combination, where n_1 indicates the group ID and n_2 is the iteration number.

Algorithm 6: Vertex Grouping

```
1  it ← 1
2  joining ← true
3  while (joining)
4      joining ← false
5      parallel for each GPU thread
6          gid ← GPU thread's global ID
7          nbr ← selected_neighbor[gid]
8          nbr_group_ID ← get_group_ID(group_comb[nbr])
9          cur_group_ID ← get_group_ID(group_comb[gid])
10         if cur_group_ID > nbr_group_ID then
11             max_group_comb ← get_group_comb(cur_group_ID, it)
12             atomicMax(&group_comb[nbr], max_group_comb)
13             joining ← true
14         else if cur_group_ID < nbr_group_ID then
15             max_group_comb ← get_group_comb(nbr_group_ID, it)
16             atomicMax(&group_comb[gid], max_group_comb)
17             joining ← true
18     it++
```

The thread then creates a new group combination using the larger group ID and the iteration number (line 11 or 15) and uses *atomicMax* to update the group combination of either its vertex or its selected neighbor, whichever is smaller (line 12 or 16). Since multiple threads may attempt to update the same vertex's group combination concurrently, we use *atomicMax* to ensure these updates are performed atomically. Finally, threads that update the group combination sets *joining* to true (line 13 or 17), indicating that there are still vertices joining other groups and that threads need to continue updating group combinations. Once all vertices have finished joining groups and group combinations no longer change, all vertices in the same group will have group IDs equal to the vertex ID of their group leader.

Figure 2.2 shows an example of our grouping algorithm converging in 3 iterations. In this example, v_4 is the group leader since it has the largest vertex ID. v_3 is closest to the group leader, resulting in a smaller group ID compared to v_1 and v_2, which are farther away from the group leader.

Algorithm 7 shows our grouping algorithm. Initially, each vertex is in its own group. We assign each vertex to a GPU thread, with each thread responsible for updating the group ID of its assigned vertex and its selected neighbor to the largest group ID between them. Since multiple threads may attempt to update the same vertex's group ID concurrently, we use *atomicMax* to ensure these updates are performed atomically. This grouping algorithm is performed iteratively until all vertices' group IDs converge, meaning no vertex has changed its group ID in the previous iteration. Once it converges, all vertices in the same group will have a group ID equal to the highest vertex ID (i.e., group leader) within that group.

After grouping vertices, some groups become significantly larger than others, making it difficult to achieve a balanced partition at a later initial partitioning stage. One simple solution is to randomly divide each large group into smaller subgroups, with each subgroup having a maximum size of s. However, this approach can lead to poor partition quality because vertices that are far apart may end up in the same subgroup. Coarsening these distant vertices together can distort the original graph structure.

To address this issue, our algorithm ensures that vertices closer to the group leader have smaller group IDs. Specifically, vertices closer to the group leader will converge faster and adopt the smallest group ID, while more distant vertices will take longer to converge and will have larger group IDs. Figure 2.2 shows an example of our grouping algorithm converging in 3 iterations. In this example, v_4 is the group leader since it has the largest vertex ID. v_3 is closest to the group leader, resulting in a smaller group ID compared to v_1 and v_2, which are farther away from the group leader. In our algorithm, each vertex's group ID is an eight-byte data type, where the first four bytes store the vertex's group ID, and the last four bytes indicate the iteration during which the group ID was last updated. This setup allows us to update the group ID with a single atomic operation.

Algorithm 7: Balanced Subgroup Breakdown

```
1 parallel for each group
2     gid ← GPU thread's global ID
3     $\mathtt{i^{th}}$ ← the $\mathtt{i^{th}}$ vertex in the group        ▷ position within the group
4     sub_group_start ← floor($\mathtt{i^{th}}$ / $s$)
5     sub_group_id ← v_id[sub_group_start × $s$]
6     group_id[gid] ← sub_group_id
```

Balanced Subgroup Breakdown

After computing the group combinations, we sort the vertices in ascending order based on their group combinations, so that vertices belonging to the same group are placed together. Within each group, vertices are ordered according to their distance from the group leader (i.e., vertices farther from the group leader have larger group IDs). We then divide the vertices into subgroups based on their position within the group, with each subgroup having a maximum size of s. We maintain an array

v_id to record each vertex's vertex ID in the same order as they appear in $group\_comb$. Algorithm 7 shows our balanced subgroup breakdown algorithm. For each group, we assign each vertex in the group to a GPU thread. The thread first identifies the index of its vertex within the group (line 3). The thread then computes the subgroup for the vertex by dividing the index by s (line 4). To assign the subgroup a new group ID and ensure that no two subgroups share the same ID, the thread locates the first vertex within its subgroup and uses this vertex's vertex ID as the subgroup ID (line 5). Finally, the thread updates the group ID accordingly (line 6).

Sequence-based Refinement

The goal of a refinement algorithm is to minimize the number of cut hyperedges by reducing the number of hyperedges that span multiple partitions. This is achieved by moving vertices among different partitions. We define a *vertex_move* as $m_u^{p_{cur}, p_{dst}}$ that represents moving the vertex u from the current partition p_{cur} to the destination partition p_{dst}. We then define the gain, $gain(u, p_{dst})$, of a vertex move $m_u^{p_{cur}, p_{dst}}$ as follows:

$$\sum_{e \in E : u \in e} W_e \times \{\delta(num\_pins(e, p_{cur}) = 1) - \delta(num\_pins(e, p_{dst}) = 0)\},$$

where $\delta(condition)$ is an indicator function that returns 1 if the condition is true and 0 otherwise. The function $num\_pins(e, p_i)$ returns the number of pins of edge e that are located in partition p_i. The gain is computed by summing all hyperedges e incident to u. For each e:

- The first term $\delta(num\_pins(e, p_{cur}) = 1)$ checks if e has only one pin left in p_{cur}. If the condition is true, then moving u from p_{cur} will result in a positive contribution to the gain, as it reduces the number of partitions e spans.

- The second term $\delta(num\_pins(e, p_{dst}) = 0)$ checks if e has no pins in p_{dst}. If the condition is true, then moving u to p_{dst} will result in a negative contribution to the gain, as it increases the number of partitions e spans.

Parallel refinement can make each vertex's gain inconsistent due to the concurrent movement of adjacent vertices [18]. To ensure correct gains, G-kway [18] identifies an independent set of vertices to move in parallel. However, such a strategy largely reduces the available parallelism for hypergraphs. Since a hyperedge can connect to many vertices, the size of an independent set in a hypergraph is often small. To address this problem, we propose a sequence-based refinement algorithm. We first find a sequence of vertex moves with positive gain in the descending order. Next, we update the gain of each vertex move in the sequence by assuming previous vertex moves are already applied. This strategy allows us to update the gain of a vertex move based on its neighbors' latest partition locations, eliminating the need for an independent set. Since a vertex move may have negative gains after updating, we accumulate the gains to identify the best subsequence of vertex moves that yields the largest gain while maintaining balanced partitions.

Our sequence-based refinement algorithm consists of two steps: *Sequence of vertex moves finding* and *Best subsequence selection*.

Sequence of Vertex Moves Finding

In this step, our goal is to find vertices with positive gains and store them in a sequence of vertex moves *seq_vertex_moves*. To avoid redundant gain computations, we record the adjacent partitions of each vertex u (i.e., partitions where u has neighbors) using a 64-bit data type adj_par. In *adj_par*, each bit represents a partition and its value is either 1 or 0. Specifically, if p_i is adjacent to u, the value of the i^{th} bit in u's *adj_par* is 1;

Algorithm 8: Sequence of Vertex Moves Finding

```
parallel for each GPU warp
    warp_id ← GPU warp's global ID
    lane_id ← GPU thread's ID within a warp
    v_start ← warp_id * 32
    v_end ← v_start + 32
    for each $v \in \{v\_start \dots v\_end\}$
        $p_{cur}$ ← u's current partition
        max_gain ← 0
        max_gain_p ← $\emptyset$
        for each $p \in v$'s adj_par
            e_id ← lane_id$^{th}$ $e \in u$
            e_gain ← $W_e \times \{\delta(\text{num\_pins}(e, p_{cur}) = 1)$ - $\delta(\text{num\_pins}(e, p) = 0)\}$
            sum_gain ← __reduce_add_sync(0xffffffff, e_gain)
            if lane_id == 0 && sum_gain > max_gain then
                max_gain ← sum_gain
                max_gain_p ← p
            __syncwarp()
        if lane_id == 0 && max_gain > 0 then
            pos ← atomicAdd(seq_size, 1)
            seq_vertex_moves[pos] ← $m_v^{p_{cur}, max\_gain\_p}$
        __syncwarp()
```

otherwise, it is set to 0. This information is updated as vertices are moved. We only calculate gains for the adjacent partitions of u since moving a vertex to a non-adjacent partition will not result in a positive gain. This strategy largely reduces unnecessary gain calculations.

To find vertices with positive gains, we assign each GPU warp 32 consecutive vertices. All threads in a warp calculate the gain of a vertex u for one of its adjacent partitions at a time. Since each vertex can have a different number of adjacent partitions and hyperedges, the time for computing a vertex's gain is different. In our kernel, if a warp finishes processing a vertex quickly, it can immediately proceed to the next one. This approach increases warp occupancy, ensuring enough active warps to keep the GPU cores busy. Algorithm 8 shows our sequence of vertex moves finding algorithm. In each warp, all threads first fetch 32 consecutive vertices (lines 4-5), and they calculate the gain of a vertex v to one of its adjacent partitions P at a time. Specifically, each thread fetches one of u's hyperedges e and calculates the gain for e (lines 11-12). We use *__reduce_add_sync* to efficiently sum up the gain of each e computed by each thread (line 13). After obtaining the total gain, the first thread in the warp (i.e., lane ID equals 0) checks if the total gain is greater than the current maximal gain (line 14). If it is true, the thread updates the current maximal gain and the associated partition (lines 15-16). Then, all threads continue processing v's next adjacent partition. Once threads have processed all of v's adjacent partitions, the first thread checks if v has a maximal gain greater than zero (line 18). If it does, the thread atomically increments a variable indicating the current sequence size seq_size to get a position in *seq_vertex_moves* and inserts the vertex move into the sequence (lines 19-20).

After finalizing the sequence of vertex moves, we need to select a subsequence of them such that applying those vertex moves yields the largest gain while still satisfying the balance constraint. However, finding the optimal subsequence encounters the problem of exponential enumeration, as each vertex move must be considered for selection or not. Specifically, identifying a subsequence of size k requires evaluating 2^k possible combinations. To address this problem, we sort each vertex move by gain and select vertex moves in descending order. This selection allows us to move vertices with larger gains first, thus maximizing the improvement in cut size.

Best Subsequence Selection

The goal of this step is to select the best subsequence of vertex moves to move. We first update the gain of each vertex move by assuming that

Algorithm 9: Gain Updating

```
1  assign a $m_u^{p_{cur}, p_{dst}}$ to a GPU warp
2  parallel for each GPU warp
3      lane_id ← GPU thread's ID within a warp
4      u_move_order ← move_order[u]
5      gain ← 0
6      foreach u's hyperedge e
7          pin_id ← lane_id^th pin in e
8          pin_move_order ← move_order[pin_id]
9          pin_par ← pin_move_order < u_move_order ? seq_vertex_moves
             [pin_move_order].dst : seq_vertex_moves [pin_move_order].cur
10         num_pins_cur ← number of pin_pars are $m_u^{p_{cur}, p_{dst}}$.cur
11         num_pins_dst ← number of pin_pars are $m_u^{p_{cur}, p_{dst}}$.dst
12         if lane_id == 0 && num_pins_dst == 0 then
13             gain − = $W_e$
14         if lane_id == 0 && num_pins_cur == 1 then
15             gain + = $W_e$
16         __syncwarp()
17     if lane_id == 0 then
18         seq_vertex_moves[u_move_order] .gain ← gain
```

previous vertex moves are all applied. The gain of a vertex move is updated by computing its neighbors' current partitions based on their order in the sequence of vertex moves. To allow each thread to quickly access a vertex move's order without searching the entire sequence, we maintain a *move_order* array of length $|V|$, where the index of vertex u records u's order in the sequence. Algorithm 9 presents our parallel gain updating algorithm. We assign each vertex move $m_u^{p_{cur}, p_{dst}}$ to a GPU warp, where all threads in the warp update the gain of $gain(u, p_{dst})$, by computing the gain of each hyperedge e of u at a time. Each thread first fetches one of e's pins pin and its order *pin_move_order* from *move_order* to a thread-local variable (lines 7-8). Each thread then finds pin's current partition *pin_par* by comparing the order of pin and u (line 9). If pin's order is less than u's, meaning pin moves before u, then by the time we are moving u, pin

should already be in its destination partition. Conversely, if pin's order is greater than u's, meaning pin moves after u, then by the time we are moving u, pin should still be in its current partition. Based on the order of pin, each thread determines its pin's current partition and counts the number of pins in $m_u^{p_{cur},p_{dst}}$'s current partition p_{cur} (line 10). To efficiently calculate the number of pins in the p_{cur}, we use *__ballot_sync* to identify threads whose $\mathrm{pin\_par}$ matches p_{cur} and *__popc* to count those threads. The number of pins in the destination partition is computed similarly (line 11). Finally, the first thread in the warp checks if e has no pins in p_{dst}. If this condition is met, e results in a negative gain, and the thread decrements the gain by W_e (lines 12-13). Additionally, the thread checks if e has only one pin left in p_{cur}. If this condition is met, e results in a positive gain, and the thread increments the gain by W_e (lines 14-15).

After updating the gain of each vertex move, some vertex moves may have negative gains. To find a subsequence with the largest total gain, we employ a GPU scan algorithm to accumulate the gains of the sequence of vertex moves and store them in the array *accum_gains*. This allows us to calculate the total gains that we can obtain if we apply all the vertex moves in the subsequence. For example, the j^{th} element in *accum_gains* stores the total gain obtained by applying the vertex moves from the first to the j^{th} (i.e., a subsequence from the first to the j^{th} vertex move). We can then find the subsequence with the largest gain.

Next, we use the same strategy to compute the partition balance result. We maintain k different $\mathrm{del\_wgt}_i$ arrays for k partitions, each with a size equal to the number of vertex moves in the sequence. Each element in $\mathrm{del\_wgt}_i$ records the weight change of the i^{th} partition after applying a vertex move. Specifically, the value in $\mathrm{del\_wgt}_1[0]$ records the first partition's weight change after applying the first vertex move. We then use a GPU scan on each $\mathrm{del\_wgt}_i$ to get the total partition weight change for each partition. After scans, the j^{th} element in $\mathrm{del\_wgt}_i$ stores the total partition weight change for partition i from applying the first to the j^{th} vertex moves. Based on the total partition weight changes, we can compute if the partition will be balanced after applying vertex moves in the subsequence. Using the accumulated gains *accum_gains* and the result of each del_wgt_i, we can find the longest subsequence where the total gain is maximized and the resulting partition remains balanced. We then apply all vertex moves in this subsequence in parallel.

2.4 Experimental Evaluation

We evaluated the performance of HyperG on 18 industrial circuit graphs derived from the ISPD98 VLSI Circuit Benchmark Suite [33]. Since the original graphs are small (a few thousand vertices), we expanded the circuit graphs 100–1000 times larger with random vertex and edge insertions to demonstrate the advantage of GPU parallelism. We implemented HyperG using C++17 and CUDA 12.0 and compiled it with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 16 Intel i7-11700 CPU cores at 2.50 GHz and 128 GB RAM. Our GPU is A6000 with 48 GB global memory.

Baselines

We consider two state-of-the-art hypergraph partitioners as our baseline, hMetis [14] (sequential) and Mt-KaHyPar [2] (multithreaded). For all experiments, we set the imbalance ratio to 3%. In the coarsening stage, we set the maximum size of each subgroup to four and terminate the coarsening algorithm when the number of vertices drops below $160 \times k$ or less than 95% of the vertices can be coarsened in the previous coarsening level. We use the default settings for both hMetis and Mt-KaHyPar.

Overall Performance Comparison

Table 2.1 compares the overall runtime and cut size results among hMetis, Mt-KaHyPar, and HyperG at $k = 2$. On our machine, Mt-KaHyPar saturates at about 10–16 threads. Hence, we report its results under 16 threads. In terms of runtime, HyperG outperforms hMetis and Mt-KaHyPar across nearly all circuit graphs, with an average speedup of $133\times$ and $4.1\times$, respectively. The largest speedups we observe are $177\times$ over hMetis and $6\times$ over Mt-KaHyPar. For the smallest graph (circuit08), the runtime of HyperG is slightly slower than Mt-KaHyPar ($0.9\times$), which is due to the limited data parallelism exhibited by this graph. For the largest graph (circuit17), hMetis failed to finish due to a memory error. Regarding cut size, HyperG always achieves comparable values with both hMetis and My-KaHyPar.

We attribute this promising performance to our efficient balanced group coarsening algorithm, which groups vertices into subgroups of similar sizes and coarsens all vertices within these subgroups. This algorithm largely reduces the number of coarsening levels by efficiently coarsening many vertices at each level while ensuring that the resulting coarsened vertices have similar sizes. This balance in vertex weights helps the initial partitioning stage find a good initial solution. Additionally, our sequence-based refinement algorithm correctly computes the gains for a sequence of vertex moves. This strategy allows us to identify the best subsequence of vertex moves to apply in parallel, reducing the number of refinement steps while ensuring high partitioning quality.

Runtime Analysis

Figure 2.3 shows the speedup of HyperG over Mt-KaHypar (16 threads) and hMetis at $k = \{2, 4, 8, 16, 32, 64\}$ on four circuit graphs, circuit01, circuit02, circuit06, and circuit09. We chose these four circuit graphs because

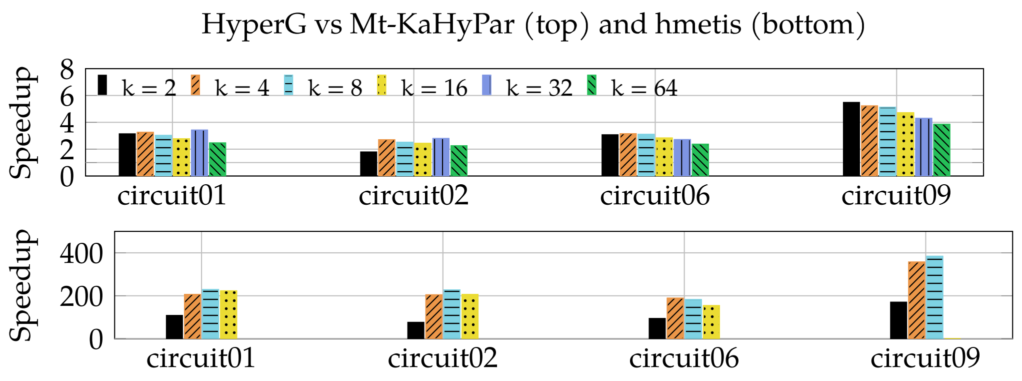


Figure 2.3: The speedup of HyperG over Mt-KaHyPar (top) and hMetis (bottom) at different k. In cases where hmetis fails to partition the circuit graph, the results are left blank.

hMetis can finish the partitioning in most k during our settings.

Regardless of k, HyperG is always faster than hMetis and Mt-KaHyPar. For example, with $k = 2$, HyperG achieves up to 5.5× and 380× speedup over Mt-KaHyPar and hMetis, respectively; With $k = 64$, HyperG is up to 4× faster than My-KaHyPar, whereas hMetis fails to finish due to a memory error. We attribute this to our balanced group coarsening, where we largely reduce the number of vertices at each coarsening level. Moreover, our sequence-based refinement algorithm can move many vertices in parallel, thus significantly reducing the time spent in the refinement stage.

We also observe the impact of k on the performance of HyperG compared with hMetis and Mt-KaHyPar. As k increases, the runtime of all partitioners increases. However, the runtime of hMetis increases much faster than HyperG and Mt-KaHyPar. For example, on circuit09, when k goes from 2 to 4, HyperG and My-KaHyPar become 8.92% and 3.64% slower, respectively, while hMetis becomes 128.5% slower.

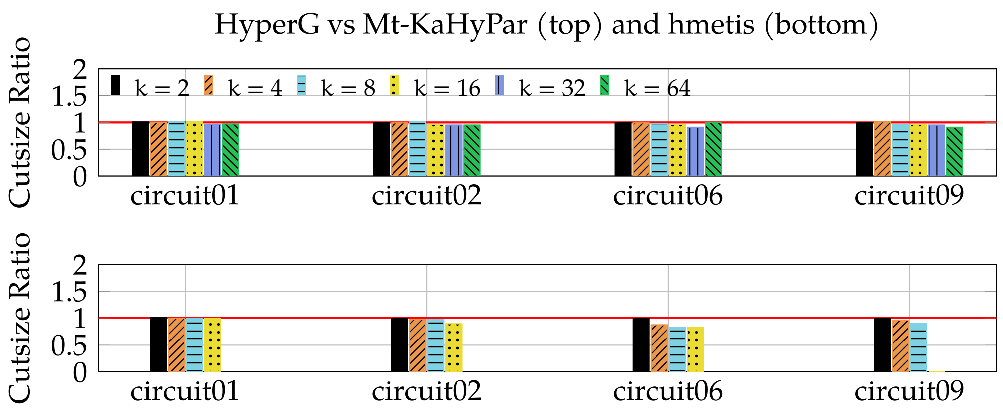


Figure 2.4: The cut size ratio of HyperG over Mt-KaHyPar (top) and hMetis (bottom) at different k. In cases where hmetis fails to partition the circuit graph, the results are left blank.

Cut Size Analysis

Figure 2.4 shows the cut size improvement ratio of HyperG over Mt-KaHyPar (16 threads) and hMetis at $k = \{2, 4, 8, 16, 32, 64\}$ on four circuit graphs, circuit01, circuit02, circuit06, and circuit09. For most circuit graphs, HyperG can produce partitions with comparable quality to hMetis and Mt-KaHyPar. Compared to mt-KaHyPar, HyperG can partition all graphs with $\leqslant 5\%$ cut size differences. We attribute this to our balanced group coarsening algorithm, which leads to a good initial partition. Additionally, our sequence-based refinement algorithm can correctly compute the improvement in cut size for a sequence of vertex moves and select the best subsequence of them to move. However, this selection can sometimes trap us in local minima. Therefore, in sequential partitioning, hMetis, which iteratively finds the best vertex to move, can result in a more refined and optimized partition. Yet, HyperG can still find partitions with less than 5% difference in quality from hMetis for 60% of the instances.

In terms of the impact of k on partition quality, regardless of k, HyperG always finds a very similar cut size as Mt-KaHyPar. Compared to hMetis, when k is small (e.g., 2 and 4), HyperG can achieve similar cut size quality in nearly all graphs. However, as k increases, hMetis can sometimes find a better cut size. Yet, for larger k values (e.g., 32 and 64), hMetis fails to partition circuit graphs due to memory issues.

Scalability Analysis

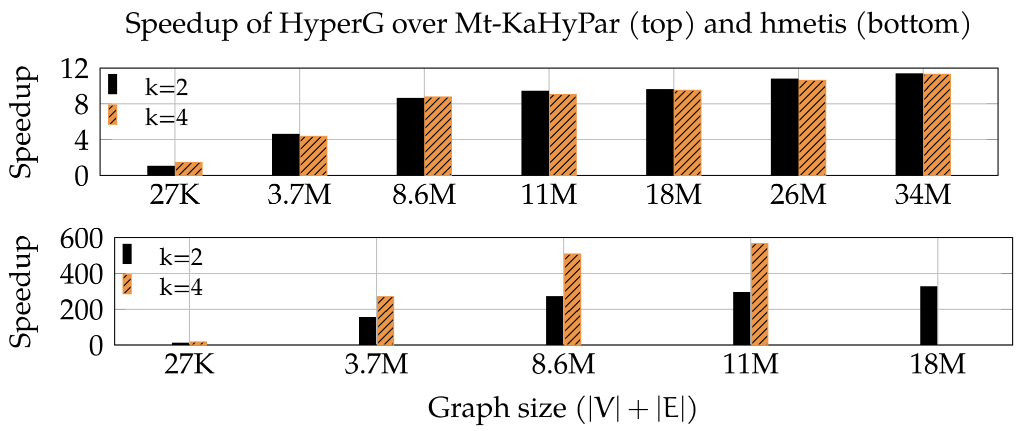


Figure 2.5: The speedup of HyperG over Mt-KaHyPar (top) and hMetis (bottom) for varying circuit graph sizes modified from ibm01 at $k = 2$ and $k = 4$. hmetis fails to partition the circuit graph with size larger than 18M.

Figure 2.5 shows the speedup of HyperG over Mt-KaHyPar and hMetis when partitioning circuit graphs of varying sizes at $k = 2$ and $k = 4$. We choose only these two k values because hMetis fails to partition the graphs at larger k values. We generate different circuit graph sizes ($|V| + |E|$) by randomly inserting vertices and edges to enlarge ibm01 [33] from 27K to 34M. When the graph size is small, the speedup is subtle. For example, at 27K, the speedup of HyperG over Mt-KaHyPar and hMetis is only 1.2× and 10×, respectively. However, as the graph size goes beyond 3M vertices, HyperG shows a significant performance advantage over the CPU-based parallel Mt-KaHyPar and sequential hMetis. Moreover, hMetis fails to partition graphs beyond 18M. For example, for the largest graph (34M), HyperG achieves an 11× speedup over Mt-KaHyPar, while hMetis fails to partition the graph. The speedup of HyperG continues to increase as the graph turns larger, showing the advantage of GPU acceleration for large-scale graph partitioning.

Table 2.2: Comparison between two GPU-accelerated partitioners, HyperG (hypergraph) and G-kway (non-hypergraph) at $k = 2$. G-kway requires an extra transformation step.

Benchmark	HyperG		G-kway		
	Cut size	Part. time (s)	Cut size	Trans. time (s)	Part. time (s)
circuit01	1,498	0.770	3,533	7.152	0.125
circuit02	1,572	1.692	3,493	20.310	0.372
circuit03	1,665	0.908	3,549	8.180	0.165
circuit04	1,699	1.567	3,038	7.63	0.145

Comparison with Graph Partitioners

Table 2.2 compares the cut size and runtime results between two GPU-accelerated partitioners, HyperG (hypergraph) and G-kway (non-hypergraph) [18] at $k = 2$. Since G-kway is a non-hypergraph partitioner, it requires an extra transformation of hypergraphs into non-hypergraphs. To transform a hypergraph to a non-hypergraph, we use a classical clique-expansion method where each hyperedge is replaced with a clique [14, 28]. In all benchmarks, HyperG consistently produces better cut sizes than G-kway. This is because HyperG can work directly on the hypergraph, preserving the original multi-vertex relationships and effectively minimizing the cut size. However, G-kway introduces many more edges by transforming the hypergraph into a non-hypergraph, which leads to an increased edge count and a loss of the original hypergraph structure.

2.5 Conclusion

In this chapter, we have introduced HyperG, a GPU-accelerated hypergraph partitioner to achieve significant runtime improvement that was previously out of reach with a CPU. HyperG introduces an innovative balanced group coarsening and a sequence-based refinement algorithm to accelerate both the coarsening and uncoarsening stages. Experimental results have shown the promising performance of HyperG over state-of-the-art CPU-parallel hypergraph partitioners on large industrial circuits.

In this work, Wan Luan Lee was the primary contrib- utor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and over- sight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Table 2.1: Overall comparison of runtime (second) and cut size among hMetis (sequential), Mt-KaHyPar (16 threads), and HyperG at $k = 2$. The last two columns show the speedup of HyperG over hMetis and Mt-KaHyPar. Best cut sizes are in bold.

Hypergraph benchmark			hmetis (Sequential)		Mt-KaHyPar (16 threads)		HyperG		Speedup vs	
Name	# Vertices	# Edges	Time (s)	Cut size	Time (s)	Cut size	Time (s)	Cut size	hmetis	Mt-KaHyPar
circuit01	2,639,664	2,920,977	83.359	**1,480**	2.415	1,513	0.770	1,498	108.3×	3.1×
circuit02	5,076,659	5,072,256	246.654	**1,570**	5.784	1,597	1.692	1,572	145.8×	3.4×
circuit03	3,215,904	3,808,739	116.118	1,666	3.368	1,666	0.908	**1,665**	127.9×	3.7×
circuit04	3,273,333	3,804,430	114.703	**1,686**	3.440	1,693	1.567	1,699	73.2×	2.2×
circuit05	5,898,747	5,717,646	243.087	815	6.608	828	2.134	**814**	113.9×	3.1×
circuit06	5,817,142	6,233,854	235.252	1,708	7.179	1,692	1.351	**1,691**	174.1×	5.3×
circuit07	5,648,898	5,918,391	208.098	**1,744**	6.717	**1,744**	1.318	1,745	157.9×	5.1×
circuit08	2,001,051	1,970,007	67.163	**853**	1.734	854	1.896	**853**	35.4×	0.9×
circuit09	4,965,735	5,663,886	175.453	**1,784**	5.652	1,794	1.031	1,794	170.2×	5.5×
circuit10	6,179,181	6,692,444	251.446	**1,804**	7.855	1,807	1.526	1,808	164.8×	5.1×
circuit11	5,856,314	6,760,682	210.619	**1,802**	7.155	**1,802**	**1.197**	**1,802**	176.0×	6.0×
circuit12	3,767,028	4,093,720	141.953	**1,801**	4.237	1,806	1.956	1,811	72.6×	2.2×
circuit13	3,620,557	4,285,638	122.969	1,836	4.322	**1,835**	0.998	**1,835**	123.2×	4.3×
circuit14	4,163,763	12,487,976	176.301	1,848	6.194	1,848	1.197	**1,802**	147.3×	5.2×
circuit15	5,166,175	5,347,020	264.711	**1,859**	8.505	**1,862**	2.136	**1,859**	123.9×	4.0×
circuit16	7,889,812	8,172,064	338.495	1,868	10.840	**1,866**	2.005	**1,866**	168.8×	5.4×
circuit17	11,686,185	11,943,603	N/A	N/A	18.168	**1,857**	3.225	1,861	N/A	5.6×
circuit18	7,371,455	7,067,200	122.969	**1,852**	9.899	1,853	2.191	**1,852**	177.1×	4.3×
Average									133.0×	4.1×

3 iG-kway: Incremental k-way Graph Partitioning on GPU

Recently, researchers have leveraged the GPU to accelerate graph partitioning to a new performance milestone. However, existing GPU-accelerated graph partitioners are limited to full graph partitioning and do not anticipate incremental updates. Incremental partitioning is integral to many optimization-driven CAD applications, where a circuit graph is repartitioned iteratively as it undergoes incremental modifications during the evaluation of optimization transforms. To unlock the full potential of GPU-accelerated graph partitioning, we introduce *iG-kway*, an incremental k-way graph partitioner on GPU. iG-kway introduces an incrementality-aware data structure to support graph modifications directly on the GPU. Atop this data structure, iG-kway introduces an incremental refinement kernel that can efficiently refine affected vertices after the graph is incrementally modified, with minimal impact on partitioning quality. Experimental results show that iG-kway achieves an average speedup of 84× over the state-of-the-art G-kway, with comparable cut sizes.

3.1 Introduction

Graph partitioning is important for the design of efficient computer-aided design (CAD) algorithms because it allows an algorithm to break down a circuit graph into smaller and manageable pieces. However, as circuit graphs continue to grow in size, graph partitioning becomes increasingly time-consuming. For example, the sequential graph partitioner, metis [10], can take several minutes to partition just one million-node graph [34], and the runtime keeps increasing as the graph size becomes larger. To reduce the runtime, researchers have proposed various parallel graph partitioning algorithms [35, 1, 36, 37, 38, 39, 40, 41, 42, 7]. For example, mt-metis [1] parallelized multi-level graph partitioning using multi-core CPUs, but the speedup is limited to only 8–16 CPU threads [18]. More recently, G-kway [18], Jet [34], and GKSG [17, 43] explored data parallelism from different stages of graph partitioning and offloaded time-consuming tasks (e.g., coarsening, refinement) to the GPU, achieving significant speedup for large graphs.

In addition to partitioning an input graph once, which we refer to as *full graph partitioning* (FGP), *incremental graph partitioning* (IGP) is a critical enabler for many CAD applications that incorporate graph partitioning in a loop. For instance, the multi-input RTL simulator [12, 44] counts on iterative IGP to discover an optimal task graph for heterogeneous scheduling; Similarly, a timing-driven optimizer also relies on iterative IGP to enhance the runtime performance of a timing analyzer [11]. In these applications, when the graph is incrementally modified, the partitioner must quickly refine the partitioning result to maintain a reasonable turnaround time across thousands or even millions of incremental iterations. As shown in Figure 3.1, IGP can save a significant amount of time compared to FGP. Without IGP, we cannot fully unlock the benefit of graph partitioning.

IGP has been studied in prior works [45, 46, 47, 48, 49], with a core focus on refining small subgraph regions affected by graph modifications, rather than performing a full re-partitioning from scratch. However, these works are largely limited to CPU architectures and can become inefficient when handling large graphs or when the affected regions are large. Inspired by the recent success of GPU-accelerated FGP [18], we believe that a GPU can also enhance the performance of IGP due to the large amount of data parallelism exhibited by graph partitioning. More importantly, as more CAD applications begin leveraging GPU acceleration [50, 51, 52, 12, 44, 53, 54, 55, 56**?** , 57, 58, 59, 60, 61, 62, 63], there is an increasing need to re-target time-consuming CPU tasks to GPU. For instance, a GPU-accelerated IGP will further speed up the incremental task graph optimization process of [12] (which takes several hours) while reducing the cost of moving and converting graph data between CPU and GPU during iterative IGP.

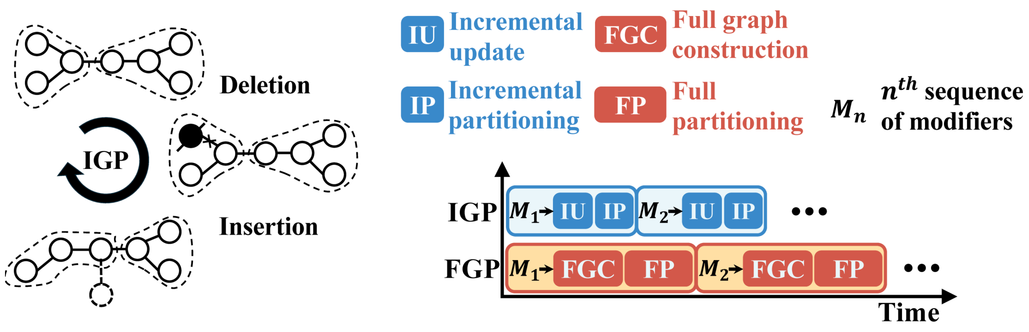

Deletion
IGP
Insertion
IU Incremental update
FGC Full graph construction
IP Incremental partitioning
FP Full partitioning
M_n n^{th} sequence of modifiers
IGP
FGP
M_1 IU IP M_2 IU IP
M_1 FGC FP M_2 FGC FP
Time

Figure 3.1: Incremental graph partitioning (IGP) and its runtime advantage over full graph partitioning (FGP).

However, designing a GPU-parallel incremental graph partitioner is very challenging. First, existing GPU partitioners [18, 17, 34] count on *static 1D arrays*, such as the compressed sparse row (CSR), to store graphs on the GPU. These static data structures make it very challenging to update graphs without rebuilding the entire arrays. Second, graph modifiers can affect vertices in different subgraph regions of different sizes. This varying number of affected vertices needs a clever strategy to balance workloads across GPU threads during incremental partitioning. Third, graph modifiers can potentially disrupt the balance of existing partitions. We need an effective GPU kernel algorithm to quickly restore partition balance while maintaining a satisfactory cut size.

While parallel IGP has been previously studied [48, 49], these efforts focused on CPU architectures. Due to the distinct performance models and memory hierarchies between CPU and GPU, we cannot directly apply these methods to GPU. For example, IOGP [48] introduced an online graph partitioning algorithm for distributed graph databases. However, IOGP focuses on optimizing data locality to minimize communication overhead in a distributed computing environment, which differs from our application focus. On the other hand, [49] formulated IGP into a two-layer linear programming (LP) proxy problem and solved it using multiple CPU threads. However, their methods require iteratively re-formulating the proxy LP problem, which is inherently sequential and cannot scale to large IGP problems.

To overcome these challenges, we introduce iG-kway, a GPU-parallel k-way graph partitioner that efficiently supports incrementality. To the best of our knowledge, this chapter presents one of the earliest research on GPU-parallel IGP, aiming to unlock the full potential of GPU-accelerated graph partitioning. We summarize our key contributions as follows:

- We introduce a GPU-aware bucket-list graph representation that can efficiently handle graph modifiers without requiring any data structure rebuilds.
- We aggregate the affected vertices in a centralized buffer and dynamically assign GPU threads to process them, ensuring balanced workloads across the GPU threads.
- We design a GPU-parallel refinement kernel algorithm that efficiently rebalances partitions by moving affected vertices to a pseudo-partition and performing incremental refinement.

We have evaluated the performance of iG-kway on industrial circuit graphs and compared our results with the state-of-the-art GPU-accelerated graph partitioner, G-kway [18]. Experimental results show that iG-kway achieves an average speedup of 84× over G-kway, with comparable cut sizes.

3.2 Problem Definition and Notation

Given an undirected graph, $G = (V, E)$, where V is a set of vertices, and E is a set of edges. Each element in E is of the form $e = (u, v)$ which represents the connection between u and v in V. For a vertex $v \in V$, we denote the weight of v by W_v, while for an edge $e \in E$, we denote the weight of e by W_e. For a vertex $v \in V$, its adjacent vertex set is denoted as $adj(v)$. Given k, if $P = \{p_1, p_2, \dots, p_k\}$ is a disjoint partition of V, we call P a k-way partition. For $v \in V$, we define $P(v) = i$ if $v \in p_i$, and its external neighbors as $adj_{ext}(v) = \{u \in adj(v) \mid P(u) \neq P(v)\}$, and its internal neighbor as $adj_{int}(v) = \{u \in adj(v) \mid P(u) = P(v)\}$. We define the cut size as $\sum_{e=(u,v)\in E, P(u)\neq P(v)} W_e$. Cut size is widely used for evaluating the quality of a partition since it represents the interconnect complexity among partitions. The partition weight of p_i is defined as $W_{p_i} = \sum_{v \in p_i} W_v$.

The goal of FGP is to find a k-way partition from scratch that satisfies the balance constraint while minimizing the cut size. The balance constraint limits the maximum weight of p_i as

$$W_{p_i} \leqslant W_{p_{\max}} = (1+\epsilon)\frac{\sum_{v\in V} W_v}{k}, \quad 0 < \epsilon \ll 1$$

where $W_{p_{max}}$ is the maximum allowable partition weight and ϵ is the imbalance ratio given by applications. Given a partitioned graph, the first goal of IGP is to apply a sequence of *graph modifiers* to the graph. Each modifier corresponds to a vertex insertion (M_u^+), a vertex deletion (M_u^-), an edge insertion ($M_{(u,v)}^+$), or an edge deletion ($M_{(u,v)}^-$). In most IGP applications [11, 64, 12], the number of modifiers is smaller than the graph size. The next goal of IGP is to efficiently refine the modified graph without starting from scratch while minimizing the cut size.

3.3 Overview of iG-kway

Figure 3.2 shows the overview of our GPU-parallel incremental k-way graph partitioner, iG-kway, which consists of two main stages: *full partitioning* (Section 3.4) and *incremental partitioning* (Section 3.5). The goal of

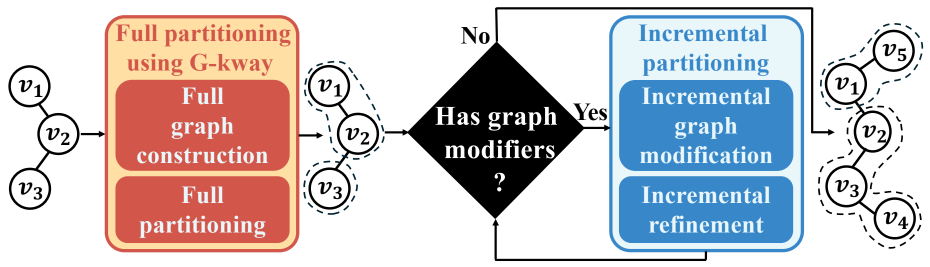
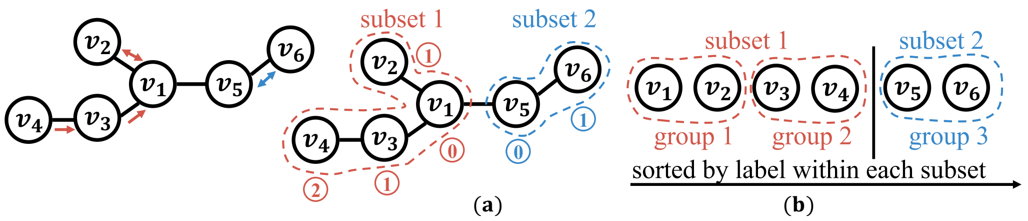


Figure 3.2: Overview of the proposed incremental graph partitioner, iG-kway.

the full partitioning is to derive a high-quality partition from the original graph, which provides a foundation for the incremental partitioner to optimize subsequently modified graphs. We utilize G-kway [18] with a new constrained coarsening strategy to achieve a high-quality partitioning result. On the other hand, the goal of incremental partitioning is to update and refine the modified graph without starting from scratch. Our incremental graph partitioning has two main stages, *incremental graph modification* and *incremental refinement*, where (1) the former introduces a bucket-list data structure that stores the input graph and supports direct modification on the GPU, and (2) the latter introduces a GPU kernel algorithm that balances and refines the partition after the graph is incrementally modified.

3.4 Full Partitioning with Constrained Coarsening

We use the state-of-the-art GPU-accelerated multilevel graph partitioner, G-kway [18], to perform full partitioning due to its high partitioning quality and fast runtime. G-kway employs a multilevel approach, iteratively coarsening the graph into a smaller representation until it reaches a manageable size, at which point the partitioning stage begins. To accelerate the coarsening process, G-kway employs a union-find-based coarsening that merges many vertices simultaneously to reduce the graph size per iteration. Here, G-kway introduces a parallel union-find algorithm to iteratively group vertices into subsets. Despite parallelism, this approach may result in imbalances in coarsened vertex weights, as each subset contains a varying number of vertices. This imbalance makes it challenging for the subsequent partitioning stage to achieve a balanced partition. Figure 3.3 (a) shows an example of G-kway's union-find-based coarsening, where G-kway groups vertices into two imbalanced subsets.



v_1 v_2 v_3 v_4 v_5 v_6
subset 1
subset 2
0
1
2
(a)
group 1
group 2
group 3
sorted by label within each subset
(b)

Figure 3.3: Examples of two coarsening methods, including (a) G-kway's union-find based coarsening and (b) our constrained coarsening. Each vertex has an arrow pointing to its selected neighbor, and its label ⓝ or ⓝ indicates the iteration when it was grouped into the subset. Vertices circled in the red dashed line will be coarsened into a coarsened vertex.

To address this issue, we divide the vertices in each subset into small, fixed-size groups of size s and merge the vertices within each group into a single coarsened vertex in parallel. This strategy reduces the graph size in each iteration by merging multiple vertices simultaneously while maintaining balanced coarsened vertex weights. To this end, a straightforward solution is randomly selecting s vertices to form smaller groups. However, this approach may result in poor partitioning quality, as vertices that are far apart could end up in the same group and be merged into a single coarsened vertex, distorting the original graph structure. For instance, in Figure 3.3 (a), randomly dividing vertices in subset 1 can result in placing distant vertices, v_2 and v_4, in the same group.

To address this problem, we modify G-kway's union-find-based coarsening algorithm by labeling each vertex with the iteration in which it joins a subset. Since vertices that are farther apart are merged into the subset in later iterations, we can sort the vertices based on their labels and divide them into groups, ensuring that vertices that are closer together are placed in the same group. Figure 3.3 (b) shows our *constrained coarsening* strategy, where vertices are first sorted by iteration number in ascending order within groups, and large subsets are divided into smaller groups of size two. In this approach, groups of similar size merge into a single coarsened vertex, producing more balanced coarsened vertex weights.

3.5 Incremental Partitioning

Bucket-list Graph Representation

Existing GPU graph partitioners [18, 17, 34] count on CSR to store graphs on a GPU. In CSR, all vertices' neighbors are concatenated into an adjacency list of size $|E|$, with an adjacency pointer recording each vertex's neighbor position. However, this statically packed data structure makes modifying the graph very challenging without rebuilding the structure. For instance, inserting an edge in the CSR data structure requires shifting all elements in the adjacency list and updating their adjacency pointers. Furthermore, updating the CSR is typically done on the CPU, which introduces additional data movement and conversion overhead between the CPU and GPU.

To overcome this challenge, we propose a bucket-list data structure that supports efficient graph modifications directly on the GPU by storing each vertex's neighbors in pre-allocated buckets. We design the buckets with 32 slots to align with the GPU warp size (32 threads), reducing thread divergence and enabling efficient intra-warp communication using CUDA warp-level primitives [65]. To avoid rebuilding the bucket-list data structure, we allocate extra buckets for each vertex to accommodate future edge insertions. The number of buckets for vertex u is determined by the following formula:

$$\left\lceil \frac{D(u)}{32} \right\rceil + \gamma$$

where $D(u)$ is the degree of u (i.e., the number of neighbors of u) and γ is the number of extra buckets per vertex (by default, iG-kway sets γ to one). To accommodate bigger graph modifications, applications can set a higher γ. Figure 3.4 shows an example of our bucket-list data structure, where each bucket contains four slots. All buckets are concatenated into a bucket-list, with a bucket pointer recording each vertex's bucket position within the list. To avoid reallocating memory when more buckets are required (e.g., due to vertex insertion), we pre-allocate a large block of memory for the bucket-list and use a pointer to track the current number of buckets.

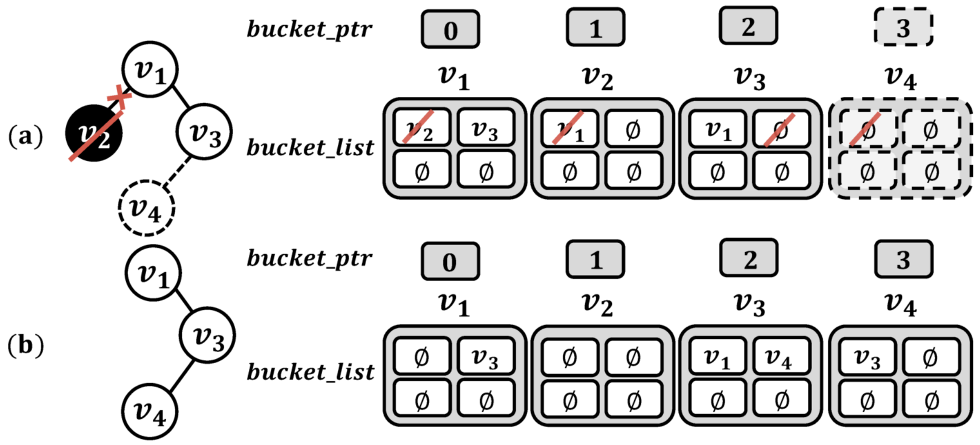


Figure 3.4: An example of our bucket-list data structure, where each vertex has a bucket, and each bucket contains four slots. Empty slots are denoted by $\varnothing$. Figures (a) and (b) show the graph before and after applying the following graph modifiers: $M^-_{v_2}$, $M^+_{v_4}$, $M^-_{(v_2,v_1)}$, $M^-_{(v_1,v_2)}$, $M^+_{(v_4,v_3)}$, and $M^+_{(v_3,v_4)}$.

Algorithm 10: Edge Insertion

```
Input: bucket_list, bucket_ptr, $M^+_{(u,v)}$, thread index in a GPU warp lane_id
1  parallel for each thread in a GPU warp
2      slot ← -1
3      bucket_cnt ← 0
4      bucket_start ← bucket_ptr[u]
5      num_bucket ← bucket_ptr[u + 1] - bucket_start
6      while slot == -1 && bucket_cnt < num_bucket
7          nbr ← bucket_list[bucket_start + bucket_cnt × 32 + lane_id]
8          if_empty ← __ballot_sync(FULL, nbr == ∅)
9          slot ← __ffs(if_empty) - 1
           // empty slot found
10         if slot ≠ -1
11             bucket_list[bucket_start + bucket_cnt × 32 + slot] ← v
12             return
13         bucket_cnt++
```

Incremental Graph Modification

Edge Modifiers

To efficiently handle $M^+_{(u,v)}$ (insert an edge (u, v)), we use a GPU warp to locate an empty slot in vertex u's buckets through fast intra-warp communication and insert vertex v to the first empty slot. Algorithm 10 presents our edge insertion algorithm. All threads in the warp first fetch the number of buckets allocated to u and the start position of its buckets from `bucket_ptr` (lines 4-5). Threads then process one bucket at a time until an empty slot is found or all buckets are checked (line 6). To efficiently locate an empty slot, each thread fetches a slot value in the bucket and uses the warp-level primitive [65], `__ballot_sync`, to simultaneously evaluate whether its slot is empty, storing the results in a bitmask (lines 7-8). If an empty slot is found, `__ffs` returns the first empty slot (line 9), and threads insert the edge into this slot and then terminate. (lines 10-12). If no slot is empty, each thread increments the bucket counter and continues looking for an empty slot in the next bucket (line 13).

We handle $M^-_{(u,v)}$ using the same strategy as Algorithm 10 except that instead of locating an empty slot, threads within the warp work simultaneously to find v in u's buckets and mark v as empty.

Algorithm 11: Vertex Insertion / Deletion

```
Input: bucket_list, bucket_ptr, vertex_status, $M_u^+$ or $M_u^-$, lane_id
1  parallel for each thread in a GPU warp
2      bucket_cnt ← 0
3      bucket_start ← bucket_ptr[u]
4      if $M_u^-$
5          vertex_status[u] ← deleted
6          num_bucket ← bucket_ptr[u + 1] - bucket_start
7      else if $M_u^+$
8          vertex_status[u] ← active
9          num_bucket ← 1
10         bucket_ptr[u + 1] ← bucket_start + num_bucket
11 while bucket_cnt < num_bucket
12     bucket_list[bucket_start + bucket_cnt × 32 + lane_id] ← ∅
13     bucket_cnt++
```

Vertex Modifiers

To delete a vertex u, a straightforward approach is to remove its buckets from the bucket-list. However, this approach can incur significant overhead, as it needs to recalculate the bucket pointer and rebuild the bucket-list. To address this problem, we use a vertex status array to track each vertex's current status (deleted or active) without removing its buckets from the bucket-list. Algorithm 11 presents our approach for handling the vertex modifier (M_u^+ and M_u^-) using a GPU warp. To handle M_u^-, threads within the same warp first mark u as deleted in the vertex status array (line 5) and cooperatively remove all of u's neighbors by marking all slots in its buckets as empty (lines 11-13). Similarly, to handle M_u^+, threads within the same warp first mark u as active in the vertex status array (line 8). Then, they assign u a single bucket and add the bucket to the end of the bucket-list by updating the bucket pointers accordingly (lines 9-10). Finally, threads initialize all slots in u's buckets as empty (lines 11-13).

Incremental Refinement

Once the graph is incrementally modified, the existing partitioning result may become invalid due to the change in the graph structure and the balance condition. For example, adding too many vertices can cause the partition to violate the balance constraint, while inserting edges may require relocating vertices to reduce the cut size. To refine a modified graph without starting from scratch, one possible approach is to use the independent-set-based refinement algorithm in G-kway [18]. However, G-kway's refinement algorithm does not account for potential imbalances that may occur after applying graph modifiers. Additionally, because it lacks IGP support, G-kway refines all vertices on the partition boundary (i.e., vertices with $\mathtt{adj}_{\mathtt{ext}} \neq 0$), which is unnecessary when only local subgraph regions are affected. To address this problem, we propose an efficient incremental refinement algorithm that restores partition balance and refines only the vertices affected by graph modifiers. Our algorithm consists of two steps, *partition balancing* and *parallel refinement*, explained below:

Partition Balancing

Incremental graph modifications can cause the partition to become imbalanced. To address this issue, we temporarily move newly added vertices to a *pseudo-partition* to prevent them from increasing the current partition weights. Additionally, to maintain the balance constraint after removing vertices, we also move the affected vertices to the pseudo-partition to reduce partition weights.

A vertex is considered affected by graph modifiers if: (1) it has been directly modified (e.g., its edges have been deleted or inserted), or (2) its neighbors have been modified, as changes in their surrounding structure may require refinement. However, not all affected vertices require refine-

Algorithm 12: Partition Balancing

```
Input: bucket_ptr, bucket_list, partition, vertex_in_pseudo, vertex_in_pseudo_size
Input: affected_vertex initialize to false
// assign a graph modifier to a GPU warp
1  parallel for each thread in the GPU warp
2      if vertex_insertion $M_u^+$
3          $partition[u] \leftarrow pseudo$
4          $pos \leftarrow$ atomicAdd($vertex\_in\_pseudo\_size$, 1)
5          $vertex\_in\_pseudo[pos] = u$
6      else if edge_insertion $M_{(u,v)}^+$ or edge_deletion $M_{(u,v)}^-$
7          $affected\_vertex[u] \leftarrow$ true ; $affected\_vertex[v] \leftarrow$ true
// assign each u in affected_vertex to a GPU warp
8  parallel for each thread in the GPU warp
9      if $partition[u] == pseudo$
10         return
11     $lane\_id \leftarrow$ thread index in the GPU warp
12     $bucket\_cnt \leftarrow 0$
13     $bucket\_start \leftarrow bucket\_ptr[u]$
14     $num\_bucket \leftarrow bucket\_ptr[u + 1]$ - $bucket\_start$
15     $cur\_par \leftarrow partition[u]$
16     while $bucket\_cnt < num\_bucket$
17         $nbr \leftarrow bucket\_list[bucket\_start + bucket\_cnt \times 32 + lane\_id]$
18         $nbr\_par \leftarrow nbr == \varnothing ? \varnothing : partition[nbr]$
19         $adj_{ext}$ += __popc(__ballot_sync(FULL, $nbr\_part$ != $cur\_par$ && $nbr$ !=
             $\varnothing$))
20         $adj_{int}$ += __popc(__ballot_sync(FULL, $nbr\_part == cur\_par$))
21         $bucket\_cnt$++
22     if $adj_{ext} > adj_{int}$ && $lane\_id == 0$
23         $pos \leftarrow$ atomicAdd($vertex\_in\_pseudo\_size$, 1)
24         $vertex\_in\_pseudo[pos] = u$
// assign u in vertex_in_pseudo to a GPU thread
25 parallel for each GPU thread
26     $partition[u] \leftarrow pseudo$
```

ment. If a vertex has as many or more adj_{int} than adj_{ext}, moving it to another partition will not reduce the cut size and may even increase it. To avoid this, we filter out such vertices and do not move them to the pseudo-partition. We use a centralized buffer, $vertex\_in\_pseudo$, to store vertices in the pseudo-partition, as they are often scattered across different parts of the graph, making it challenging to balance the workload across GPU threads. By aggregating these vertices to the buffer, we can dynamically assign GPU threads to handle them, ensuring balanced workloads and significantly increasing GPU performance.

Algorithm 12 presents our partition balancing algorithm. We use an array, $affected\_vertex$, of size $|V|$ to record whether a vertex is affected. We assign each graph modifier to a GPU warp to mark directly modified vertices as affected. Threads assigned to M_u^+ move the vertex to the pseudo-partition immediately by changing its partition in $partition$ array, which records the partition assignment of each vertex, and insert it to $vertex\_in\_pseudo$ (lines 2-5). On the other hand, threads handling $M_{(u,v)}^+$ or $M_{(u,v)}^-$ mark u and v as affected (lines 6-7). Then, we check if they can be filtered out. To do this, we assign each vertex in $affected\_vertex$ to a GPU warp. Threads assigned to vertices already in the pseudo-partition terminate early (lines 9-10), while the remaining threads process one bucket at a time, with each thread fetching a neighbor and its corresponding partition (lines 17-18). Threads efficiently calculate external and internal neighbors using the warp-level primitive `__ballot_sync` to evaluate each neighbor's partition, storing the results in a bitmask and counting them with the warp-level primitive `__popc` (lines 19-20). They then continue processing the next bucket until all are completed. Next, the first thread in the warp checks if the vertex has fewer adj_{int} than adj_{ext} (line 22). If so, it adds the vertex to $vertex\_in\_pseudo$, without immediately updating its partition in the $partition$ array (lines 23-24).

The proposed strategy in Algorithm 12 prevents data races by deferring partition updates until threads in other warps have completed their calculations. Once all affected vertices have been processed, we launch another GPU kernel to update the partitions of vertices in $vertex\_in\_pseudo$. We then move vertices with affected neighbors by assigning each vertex u in the pseudo-partition to a GPU warp. Threads in the same warp mark all neighbors of u as affected, using the same approach to filter out

Algorithm 13: Parallel Refinement

```
Input: bucket_list, bucket_ptr, partition
Input: vertex_moves, vertex_moves_size initialized to zero
Shared memory: max_nbr_p, max_nbr initialized to zero
// assign a vertex u in vertex_in_pseudo to a GPU warp
1  parallel for each thread in the GPU warp
2      lane_id ← thread index in the GPU warp
3      bucket_start ← bucket_ptr[u]
4      num_bucket ← bucket_ptr[u + 1] - bucket_start
5      while bucket_cnt < num_bucket
6          nbr ← bucket_list[bucket_start + bucket_cnt × 32 + lane_id]
7          nbr_par ← nbr == ∅? ∅ : partition[nbr]
8          if_adj_move = __any_sync(FULL, nbr_par == pseudo && nbr < u)
9          if if_adj_move ≠ 0
10             return
11         bucket_cnt++
12     for p ∈ {1...k} where $W_p < W_{pmax}$
13         bucket_cnt ← 0
14         num_nbr_in_p ← 0
15         while bucket_cnt < num_bucket
16             nbr ← bucket_list[bucket_start + bucket_cnt × 32 + lane_id]
17             nbr_par ← nbr == ∅? ∅ : partition[nbr]
18             num_nbr_in_p += __popc(__ballot_sync(FULL, nbr_part == p))
19             bucket_cnt++
20         cmp and update max_nbr and max_nbr_p in shared memory
21         if lane_id == 0
22             pos ← atomicAdd(vertex_moves_size, 1)
23             vertex_moves[pos] ← $m_u^{par,\#nbr}$
24 find the longest subsequence in parallel (Figure 3.5)
```

the neighbors whose adj_{ext} is less than adj_{int} and move the rest to the pseudo-partition.

Parallel Refinement

Once the partition is balanced and affected vertices are in the pseudo-partition, we refine them by moving each vertex to its most suitable partition in parallel. We define the most suitable partition for a vertex u as the partition to which moving u from the pseudo-partition introduces

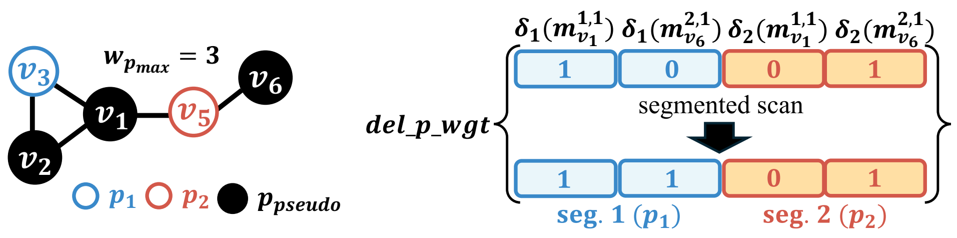


Figure 3.5: Illustration of constructing del_p_wgt to calculate the accumulated delta partition weights for two partitions, p_1 and p_2, across two vertex moves using segmented scan. All vertices have a weight equal to one.

the smallest cut size (i.e., the partition with most of u's neighbors), while maintaining the balanced partition. However, moving adjacent vertices in parallel requires costly synchronization to determine the most suitable partition accurately [18]. For example, in Figure 3.5, if vertices v_1 and v_2 move concurrently from the pseudo-partition to other partitions, v_1 may initially select either p_1 or p_2 as its most suitable partition. However, if v_2 moves to p_1, v_1 should update its choice to p_1. Without synchronization between v_1 and v_2, v_1 can select the most suitable partition incorrectly. To overcome this synchronization challenge, we move non-adjacent vertices from $vertex\_in\_pseudo$ in parallel. For each non-adjacent vertex u, we create a vertex move $m_u^{par,\#nbr}$, where par is u's most suitable partition, and $\#nbr$ is the number of neighbors in par, and insert the vertex move to the $vertex\_moves$ buffer to form a sequence of vertex moves.

Algorithm 13 presents our parallel refinement algorithm. To find non-adjacent vertices from $vertex\_in\_pseudo$, we assign each vertex u in $vertex\_in\_pseudo$ to a GPU warp. Threads in the warp cooperatively process the bucket of u one at a time, with each thread fetching a neighbor and its partition (lines 6-7). Threads then efficiently check if any thread has a neighbor in the pseudo-partition with a vertex ID less than u's vertex ID using `__any_sync` (line 8). If such a neighbor exists, threads terminate early, as u is not selected to move because its neighbor is being moved in this iteration (lines 9-10). Threads that do not find such neighbors then continue to check neighbors stored in other buckets. Next, threads with a vertex selected to move cooperatively identify the most suitable partition for u by counting u's neighbors in each partition p with a weight that does not exceed $W_{p_{max}}$. To accomplish this, each thread fetches a neighbor stored in a bucket along with its partition and uses the same strategy (i.e., warp-level primitive) as in Algorithm 12 to efficiently count neighbors belonging to p (lines 15-19). After processing all buckets, the threads update max_nbr and max_nbr_p, stored in shared memory, if either 1) the number of neighbors in partition p exceeds the current maximum max_nbr, or 2) p has the same number of neighbors as max_nbr but a lower partition weight (line 20). Finally, the first thread in the warp creates a vertex move and inserts it to the $vertex\_moves$ (lines 21-23).

After finding a sequence of vertex moves, we need to select a subsequence that, when applied, satisfies the balance constraint while introducing minimal cut size. To achieve this, we first sort the vertex moves in descending order by $\#nbr$, prioritizing vertices with strong connections with their most suitable partition. To find a subsequence that satisfies the balance constraint, we create a $delta\_p\_wgt$ array with j segments, each corresponding to a partition and with a length equal to the number of vertex moves, to record changes in partition weights if those vertex moves are applied. Each element in $delta\_p\_wgt$ records the delta partition weight of a vertex move for a partition. We define the delta partition weight of a vertex move, $m^{par,\#nbr}$ for a partition par as follows:

$$\delta_i(m_u^{par,\#nbr}) = \begin{cases} W_u, & i = par \\ 0, & otherwise \end{cases}$$

We then perform a parallel segmented scan on $delta\_p\_wgt$ to accumulate these changes for each partition.

Figure 3.5 illustrates the calculation of accumulated delta partition weights for two partitions, performed using a parallel segmented scan with two vertex moves. Each element in $delta\_p\_wgt$ records the delta partition weight for a partition after applying a vertex move. The first two elements (i.e., segment 1) correspond to partition p_1, while the last two elements (i.e., segment 2) correspond to partition p_2. For example, $delta\_p\_wgt[0]$ records the partition weight change for the first vertex move in partition p_1. We then perform a parallel segmented scan on $delta\_p\_wgt$ to accumulate these changes for each partition. After the scan, the s^{th} element in each segment holds the accumulated change in partition weight from the first to the s^{th} vertex move in each partition, representing the accumulated weight change over this subsequence. We then identify the longest subsequence of vertex moves that satisfies the balance constraint to apply as many vertex moves as possible in parallel. In this example, both vertex moves can be applied, as neither p_1's partition weight plus $delta\_p\_wgt[1]$ nor p_2's plus $delta\_p\_wgt[3]$ exceeds W_{pmax}. We then repeat the same process until all vertices are moved from the pseudo-partition.

3.6 Experimental Evaluation

We evaluated the performance of iG-kway using seven industrial circuit graphs generated by [11, 18]. Additionally, we tested iG-kway on three large non-circuit graphs (coAuthorsCiteseer, adaptive, and NLR) from the DIMACS Graph Partitioning Challenge [66] to demonstrate its applicability beyond CAD algorithms. In our experiment, we applied 100 incremental iterations based on the setting of the TAU 2015 Incremental Timing Contest [64], where each iteration involves tens to hundreds of design modifiers that randomly remove/insert vertices and edges from/into the graph.

We consider G-kway [18], a state-of-the-art GPU-accelerated k-way graph partitioner, as our baseline. To have a fair comparison with G-kway and focus on incrementality, we replace its coarsening algorithm with our constrained coarsening to achieve better performance. Furthermore, since G-kway counts on the CPU to generate a graph CSR on the GPU, we modify the graph and regenerate its CSR for each incremental iteration, then apply G-kway to partition the modified graph. Hereafter, we refer to this baseline as *G-kway*†.

We implemented iG-kway and G-kway† using C++17 and CUDA 12.0 and compiled them with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 16 Intel i7-11700 CPU cores at 2.50 GHz and 128 GB RAM. Our GPU was an A6000 with 48 GB memory. In all experiments, we set the imbalance ratio (ϵ) to 3%, the group size (s) to six, and terminated the coarsening algorithm when the number of vertices dropped below $35 \times k$ or when fewer than 90% of the vertices could be coarsened. All results are averaged over 10 runs.

Overall Performance Comparison

Table 3.1 compares the runtime and cut size between iG-kway and G-kway† at $k = 2$. To highlight the advantages of iG-kway, we break down the runtime into graph modification and partitioning. Since G-kway† does not support dynamic updates of CSR on GPU, iG-kway is always faster than G-kway† in graph modification. We attribute this runtime advantage to iG-kway's GPU-aware data structure, which stores each vertex's neighbors in buckets to rapidly respond to graph modifiers. In contrast, G-kway†'s CSR data structure is relatively static, requiring a complete rebuild to modify the graph at each iteration. Consequently, for large graphs (e.g., mem_ctrl), modification can become a significant bottleneck for G-kway†, whereas iG-kway's modification time remains consistently small, regardless of graph size.

In terms of partitioning time, iG-kway outperforms G-kway† across all graphs with an average speedup of 84×. This speedup is due to iG-kway's incremental refinement algorithm, which identifies and refines only the vertices affected by graph modifiers, eliminating the need for repartitioning required by G-kway†. Regarding the cut size, iG-kway finds a cut size comparable to G-kway† for all graphs. We attribute this to the effectiveness of our incremental refinement algorithm, which first moves vertices to a pseudo-partition to restore balance, and then moves them again to reduce the cut size. In some cases, iG-kway's incremental refinement can find a slightly better cut size than G-kway†, such as tv80, wb_dma, adaptive, and des_perf. We believe in such cases, IGP can better exploit local improvement to balance the load and minimize cuts among partitions, while FGP may be more prone to global, less granular adjustments that may overlook small local improvements.

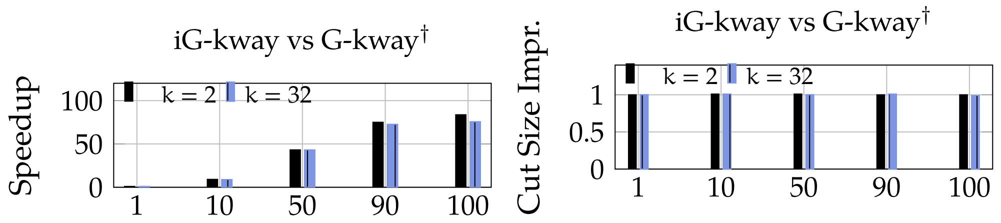


Figure 3.6: Speedup (left) and cut size improvement (right) of iG-kway over G-kway† for the usb circuit over 100 incremental iterations.

Figure 3.6 details the comparison of partitioning time and cut size over 100 incremental iterations for the circuit usb under two different values of k. At the first iteration, we do not observe a significant runtime difference between iG-kway and G-kway† since both are FGP. However, as the number of incremental iterations increases, iG-kway's runtime advantage over G-kway† becomes more pronounced. The speedup of iG-kway grows proportionally with the number of incremental iterations for both k values. In terms of cut size, iG-kway achieves a comparable value to G-kway† across all iterations (within ±3%).

Runtime and Cut Size Analysis under Varying k

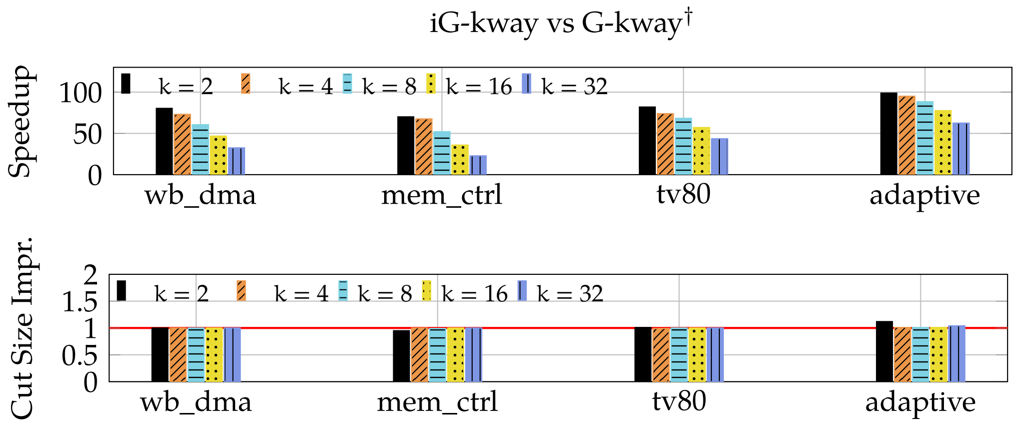


Figure 3.7: The speedup (top) and cut size improvement (bottom) of iG-kway over G-kway† at different k values. A cut size improvement above one indicates that iG-kway can find a better cut size.

Figure 3.7 shows the speedup and cut size improvement of iG-kway over G-kway† at $k = \{2, 4, 8, 16, 32\}$ on three circuit graphs (wb_dma, mem_ctrl, and tv80) and a large non-circuit graph (adaptive). Regardless of k, iG-kway is consistently faster than G-kway†. iG-kway achieves up to a 98× speedup over G-kway† at $k = 2$. However, as k increases, the speedup decreases because iG-kway needs to examine more partitions to determine a suitable partition for each affected vertex. Despite this, iG-kway still achieves up to a 62× speedup at $k = 32$. Regarding the cut size, iG-kway achieves a comparable result with G-kway† on different k. These results highlight the effectiveness and efficiency of iG-kway over G-kway† in incremental graph partitioning.

Incrementality Analysis

Figure 3.8 shows the speedup (left) and cut size improvement (right) achieved by iG-kway over G-kway† for the circuit usb across 100 incremental iterations each with varying numbers of graph modifiers (ranging from

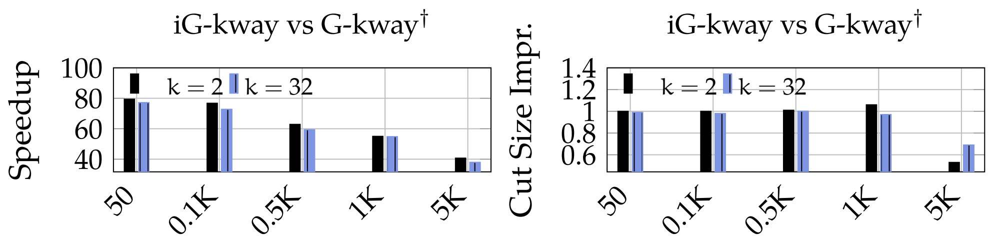


Figure 3.8: The speedup (left) and cut size improvement (right) of iG-kway over G-kway† for the usb circuit across 100 incremental iterations each with varying numbers of graph modifiers (50–5K).

50 to 5K) per iteration. When the number of graph modifiers per iteration is small, the advantage of iG-kway is more remarkable. For instance, with 50 graph modifiers, iG-kway is up to 80× faster than G-kway† while obtaining a comparable cut size. However, as the number of graph modifiers per iteration increases, the speedup decreases due to the growing number of affected vertices. More affected vertices require iG-kway to spend more time refining the partition. When the number of graph modifiers exceeds 5K per iteration, iG-kway struggles to find a partition with a decent cut size. This happens because after applying many graph modifiers (e.g., 5K×100 in this case), the graph becomes very different from its original form. This large difference makes it difficult for iG-kway to effectively optimize the graph, as its incremental refinement strategy relies on local graph structure. In such cases, applications can resort to FGP using G-kway†, especially when the number of graph modifiers reaches 50% of the graph's size. A similar strategy has been applied in GPU-accelerated incremental timing analysis [52], where a full timing update is issued after observing a big difference between the modified graph and its original form.

Analysis of our constrained coarsening

Table 3.2: Comparison of cut size between G-kway and G-kway† at k= {2, 4, 8, 16, 32}. An imbalanced partition result is marked as ✗.

	vga_lcd		coAuthorsCiteseer	
k	G-kway†	G-kway	G-kway†	G-kway
2	496	556 (↓**12.1%**)	25537	25097 (↑**1.7%**)
4	757	835 (↓**10.3%**)	42111	41354 (↑**1.8%**)
8	946	969 (↓**2.4%**)	59379	58036 (↑**2.3%**)
16	1033	1035 (↓**0.2%**)	69711	✗
32	258904	✗	78003	✗

Table 3.2 compares the cut size of G-kway† and the original G-kway to evaluate the effectiveness of the proposed constrained coarsening strategy. G-kway† consistently achieves comparable or better cut sizes than G-kway while ensuring balanced partitions across all k values. For instance, G-kway† improves the cut size by up to 12% for vga_lcd. In contrast, G-kway struggles to find balanced partitions when k is large. Without constrained coarsening, G-kway's coarsening algorithm may merge too many vertices together, creating oversized vertices that make it difficult to find balanced partitions during the later initial and uncoarsening stages. However, the proposed constrained coarsening strategy adjusts coarsening granularity by breaking down large coarsened vertices into smaller, predefined sizes. This adjustment allows G-kway† to more effectively achieve and optimize balanced partitions.

3.7 Conclusion

In this chapter, we have introduced iG-kway, a GPU-parallel incremental k-way graph partitioner. iG-kway introduces an incrementality-aware data structure to support graph modifications directly on GPU. Atop this data structure, iG-kway introduces a GPU kernel algorithm that can efficiently refine affected vertices after the graph is incrementally modified, with minimal impact on partitioning quality. Experimental results show that iG-kway achieves an average speedup of 84$\times$ over the state-of-the-art G-kway, with comparable cut sizes.

In this work, Wan Luan Lee was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and over- sight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Table 3.1: Overall comparison of runtime (modification and partitioning) and cut size between iG-kway and G-kway† at $k = 2$. All times are measured in seconds. A cut size improvement greater than one indicates iG-kway can find a better cut size.

Benchmark			Modification Time (s)		Partitioning Time (s)			Cut Size		
Name	# Vertices	# Edges	iG-kway	G-kway†	iG-kway	G-kway†	Speedup	iG-kway	G-kway†	Impr.
tv80	3,901,702	5,298,851	0.02	0.36	0.18	14.88	82.67×	**4,721**	4,774	1.01
mem_ctrl	32,445,075	42,670,885	0.11	3.37	0.58	46.07	79.43×	5,945	**5,659**	0.95
usb	139,479	180,510	0.01	0.01	0.12	10.16	84.67×	5,798	**5,701**	0.98
vga_lcd	1,869,688	23,447,678	0.07	2.13	0.38	31.27	82.29×	502	**496**	0.99
wb_dma	9,646,140	12,208,324	0.04	1.04	0.26	20.75	79.81×	**5,483**	5,489	1.00
systemcase	10,897,616	14,386,851	0.04	1.10	0.28	22.61	80.75×	4,670	**4,699**	1.00
des_perf	303,690	387,292	0.01	0.03	0.13	10.98	84.46×	**5,097**	5,150	1.01
coAuthorsCiteseer	227,320	814,134	0.01	0.03	0.13	11.20	86.15×	25,853	**25,537**	0.99
adaptive	6,815,744	13,624,320	0.03	0.97	0.51	50.12	98.27×	**1,809**	2,029	1.12
NLR	4,163,763	2,487,976	0.02	1.02	0.25	21.64	86.56×	4,611	**4,600**	1.00
Average							84.51×			1.00

4 IHYPERG: INCREMENTAL HYPERGRAPH PARTITIONING ON GPU

Recent advances in GPU-accelerated hypergraph partitioning have achieved substantial performance gains but remain limited to full partitioning. In particular, the lack of support for incrementality is a critical limitation for many CAD applications, where circuit hypergraphs iteratively undergo incremental modifications as part of optimization loops. To overcome this limitation, we present *iHyperG*, the first GPU-parallel incremental k-way hypergraph partitioner. iHyperG introduces a scalable delta-based hypergraph data structure for efficient incremental modifications on a GPU, along with an effective incremental partitioning algorithm that rebalances partitions in a single pass and refines only cut-critical vertices. Experimental results show that iHyperG achieves average speedups of 190$\times$ for modification and 83$\times$ for partitioning over the state-of-the-art GPU partitioner, while maintaining comparable partitioning quality.

4.1 Introduction

Hypergraph partitioning plays a key role in various stages of computer-aided design (CAD), including placement, routing, timing analysis, and logic simulation. For example, it helps optimize component placement by dividing the circuit into smaller, more manageable blocks while minimizing interconnections among them [4]. However, as modern circuits continue to grow in size and complexity, hypergraph partitioning has become increasingly time-consuming. For instance, the widely used sequential partitioner hMetis can take several minutes to process a circuit with only five million gates [14].

To mitigate this runtime bottleneck, several parallel partitioning strategies have been developed. Among them, Mt-KaHyPar [2] is a CPU-based hypergraph partitioner that exploits multithreading to parallelize the coarsening and refinement stages, achieving significant speedups over the sequential hMetis. More recently, HyperG [67] leverages the massive parallelism of a GPU to accelerate the partitioning process even further, reporting up to a 4× speedup over Mt-KaHyPar. Despite these runtime improvements, existing works focus only on *full hypergraph partitioning* (FHP), where the entire hypergraph is partitioned from scratch.

While FHP remains the dominant approach, many CAD applications benefit more from *incremental hypergraph partitioning* (IHP) where partitioning is integrated into iterative optimization loops. For example, a timing optimizer may repeatedly adjust cell placements to meet timing goals [11], while logic synthesis tools incrementally restructure logic cones to improve design quality [15]. In these iterative optimization workloads, each time the circuit is incrementally modified, the partitioner must rapidly refine the partitioning result to maintain a reasonable turnaround time across thousands or even millions of incremental iterations. Without IHP, the overhead of repetitive FHP can accumulate significantly, and the benefits of hypergraph partitioning cannot be fully exploited.

While incremental graph partitioning has been studied on both CPU and GPU architectures [68, 45, 46, 47, 48, 49], IHP remains largely unexplored. However, inspired by the success of the GPU-accelerated incremental graph partitioner iG-kway [68], which refines affected vertices in parallel to achieve significant runtime improvements, we believe that a GPU can similarly benefit IHP due to the substantial data parallelism inherent in hypergraph workloads. Moreover, as CAD applications increasingly adopt GPU acceleration [50, 51, 52, 12, 44, 53, 54, 55, 56**?** , 57, 58, 59, 60, 61, 62, 63, 18, 69, 70, 71], there is a growing need to retarget compute-intensive, CPU-bound tasks to a GPU. For example, in a timing optimization flow that performs incremental placement across iterations to meet timing goals [11], a GPU-based IHP framework could significantly reduce overall runtime while minimizing the overhead of frequent CPU–GPU data transfers during iterative optimization.

Although iG-kway [68] has demonstrated its effectiveness in supporting incremental graph partitioning on a GPU, its strategy cannot be directly applied to hypergraphs due to fundamental structural differences between graphs and hypergraphs. For instance, to efficiently modify graphs on a GPU, iG-kway employs a bucket-list data structure that trades higher memory usage for efficient handling of dynamic updates. However, adopting this data structure to hypergraphs results in excessive memory overhead and poor GPU scalability, as hyperedges require substantially more memory to represent complex multi-pin relationships. In addition to data structure limitations, hypergraphs also require a fundamentally different refinement strategy than graphs. For example, iG-kway refines all vertices whose incident edges have been modified, which is effective for graphs where each edge connects only two vertices. However, in hypergraphs, each hyperedge can connect many vertices and span multiple partitions. Refining all vertices that are incident to modified hyperedges can be inefficient. Therefore, we need a hypergraph refinement strategy that identifies vertices requiring updates, reducing redundant computation while preserving partition quality.

To overcome these challenges, we introduce *iHyperG*, a GPU-based k-way hypergraph partitioner that efficiently supports incrementality. To the best of our knowledge, this work represents one of the earliest efforts toward GPU-parallel IHP, aiming to fully leverage the computational power of a GPU for large-scale hypergraphs with frequent incremental modifications. We summarize our key contributions as follows:

- We introduce a delta-based hypergraph data structure that efficiently supports incremental modifications and scales to large hypergraphs.
- We propose an efficient incremental refinement strategy for hypergraphs that identifies and refines cut-critical vertices within modified hyper-

edges, reducing redundant computation while preserving partition quality.

- We implement a single-pass rebalancing algorithm that effectively restores partition balance in one pass while minimizing the increase in cut size.

- We design GPU kernels using CUDA warp-level primitives for efficient intra-warp communication, achieving high-performance incremental hypergraph modification and partitioning.

We evaluated the performance of iHyperG on industrial circuit designs and compared it against the state-of-the-art GPU-accelerated hypergraph partitioner, HyperG [67]. Experimental results show that iHyperG achieves 190$\times$ and 83$\times$ speedups in modification and partitioning, respectively, while maintaining comparable cut sizes.

4.2 Problem Definition and Notation

Given a hypergraph $H = (V, E)$, where V is a set of vertices and E is a set of hyperedges, each element $e \in E$ is a subset of V representing a multi-vertex relationship. A vertex $v \in V$ is said to be incident to a hyperedge $e \in E$ if $v \in e$; likewise, e is incident to v. This vertex–hyperedge relationship is called *incidence*. For clarity, we refer to the set of hyperedges incident to a vertex as the *vertex's incidence*, and the set of vertices incident to a hyperedge (also called *pins*) as the *hyperedge's incidence*. For a vertex v, we denote its weight as W_v, while for a hyperedge e, we denote its weight as W_e. Vertices u and v are neighbors if there exists a hyperedge $e \in E$ such that $u \in e$ and $v \in e$. Given k, if $P = \{p_1, p_2, \dots, p_k\}$ is a disjoint partition of V, we call P a k-way partition. For a hyperedge e, its connectivity $\lambda(e)$ is the number of partitions it spans (i.e., partitions containing at least one pin of e). A hyperedge introduces a *cut* if $\lambda(e) \geqslant 2$, and the cut size is the total weight of all cut hyperedges, defined as $\text{cut}(P) = \sum_{e \in E:\, \lambda(e) \geqslant 2} W_e$. For a vertex v, we define $P(v) = i$ if $v \in p_i$. The weight of the partition p_i is defined as $W_{p_i} = \sum_{v \in V : P(v) = i} W_v$.

The goal of FHP is to find a k-way partition from scratch that satisfies the balance constraint while minimizing $\text{cut}(P)$. The balance constraint limits the maximum weight of p_i as

$$W_{p_i} \leqslant W_{p_{\max}} = (1+\epsilon)\frac{\sum_{v \in V} W_v}{k}, \quad 0 < \epsilon \ll 1$$

where $W_{p_{\max}}$ is the maximum allowable partition weight and ϵ is the imbalance ratio given by applications. Given a partitioned hypergraph H, the first goal of IHP is to apply a sequence of *modifiers* to H. Each modifier is an operation on both the vertex's and hyperedge's incidence. Specifically, an insertion modifier $M^{+}_{(v,e)}$ inserts vertex v into hyperedge e's incidence and e into v's incidence, while a deletion modifier $M^{-}_{(v,e)}$ removes v from e's incidence and e from v's incidence. In typical applications, the number of modifiers is small relative to $|V|$ and $|E|$. IHP then refines the partition on the modified hypergraph without restarting from scratch, while maintaining balance and minimizing $\text{cut}(P)$.

4.3 Overview of iHyperG

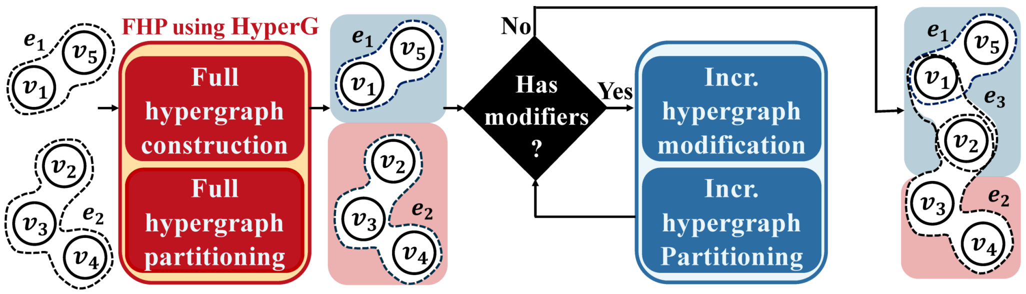

FHP using HyperG
Full hypergraph construction
Full hypergraph partitioning
Has modifiers ?
No
Yes
Incr. hypergraph modification
Incr. hypergraph Partitioning
e_1
e_2
e_3
v_1
v_2
v_3
v_4
v_5

Figure 4.1: Overview of our incremental hypergraph partitioner.

Figure 4.1 shows the overview of our GPU-parallel incremental k-way hypergraph partitioner, iHyperG, which consists of two main stages: *full hypergraph partitioning* (FHP) and *incremental hypergraph partitioning* (IHP). The goal of the FHP is to derive a high-quality partition from the input hypergraph, providing a foundation for the incremental partitioner to optimize subsequently modified hypergraphs. We use HyperG [67], a state-of-the-art GPU-accelerated hypergraph partitioner, to achieve high-quality partitions.

On the other hand, the goal of IHP is to efficiently update and refine the partition of a modified hypergraph without starting from scratch. Our IHP has two main stages, *incremental hypergraph modification* (Section 4.4) and *IHP* (Section 4.5). During the incremental hypergraph modification stage, iHyperG employs a scalable delta-based hypergraph data structure to record updated incidences of modified vertices and hyperedges, avoiding a full rebuild of the hypergraph Compressed Sparse Row (CSR) data structure. During the IHP stage, iHyperG efficiently restores balance using a single-pass rebalancing algorithm and refines only cut-critical vertices, avoiding the need to repartition the hypergraph.

4.4 Incremental Hypergraph Modification

Delta-based Hypergraph Data Structure

Existing GPU hypergraph partitioners [67, 72] store hypergraphs on the GPU using a bidirectional CSR data structure with (i) a vertex to hyperedge incidence array `V2E` and (ii) a hyperedge to vertex incidence array `E2V`. In `V2E`, each vertex's incident hyperedges are stored contiguously, and the pointer array `V2EP` records the start index of each vertex's segment in `V2E`. Similarly, `E2V` stores each hyperedge's incident vertices (pins), with `E2VP` recording the start index of each hyperedge's segment in `E2V`. While this *statically* packed data structure is well-suited for a GPU, it makes modifying the hypergraph difficult without fully rebuilding the data structure. For example, inserting a new pin into E2V requires shifting pins in E2V and updating all affected values in E2VP.

To address this challenge, one possible approach is to adopt iG-kway's [68] bucket-list data structure, which stores each vertex's incident edges in pre-allocated buckets to enable efficient graph modification on a GPU without rebuilding the entire CSR. However, this approach does not extend well to hypergraphs. Hypergraphs require maintaining both V2E and E2V, and replacing them with bucket list data structures would significantly increase memory usage and limit the size of hypergraphs that can fit on a single GPU. As a result, we introduce a scalable delta-based hypergraph data structure that efficiently supports dynamic modifications without requiring a rebuild of the original CSR structure. Our data structure consists of two components: the base hypergraph H, which stores the original incidences of vertices and hyperedges in V2E and E2V, and the delta hypergraph δH, which records the updated incidences of modified vertices and hyperedges in δV2E and δE2V.

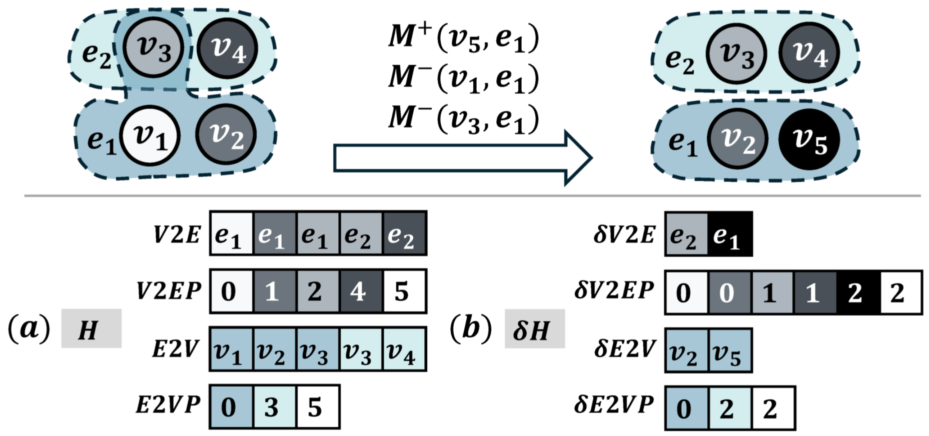


Figure 4.2: An example of our delta-based hypergraph data structure with modifiers $M^+_{(v_5,e_1)}$, $M^-_{(v_1,e_1)}$, and $M^-_{(v_3,e_1)}$, where (a) is the base hypergraph H and (b) is the delta hypergraph δH.

Figure 4.2 shows our delta-based hypergraph data structure, where

(a) is the base hypergraph H, and (b) is the delta hypergraph δH, which records the updated incidences of modified vertices v_1, v_3, and v_5, as well as hyperedge e_1, after applying modifiers $M^+_{(v_5,e_1)}$, $M^-_{(v_1,e_1)}$, and $M^-_{(v_3,e_1)}$. Vertex v_3 is disconnected from e_1, and a new vertex v_5 is inserted and connected to e_1, resulting in updated incidences $\{e_2\}$ for v_3 and $\{e_1\}$ for v_5. Additionally, vertex v_1 is disconnected from its only incident hyperedge, yielding an empty incidence. The updated incidences of the modified vertices are then recorded in δ`V2E`, with δ`V2EP` recording the start and end indices of each vertex's updated incidence within δ`V2E`. Similarly, the updated incidence $\{v_2, v_5\}$ of hyperedge e_1 is stored in δ`E2V`. The array δ`E2VP` records the start and end indices of all updated hyperedges within δ`E2V`. For unmodified vertices and hyperedges, their start and end indices in δ`V2EP` and δ`E2VP` are equal, indicating that no updated incidence is recorded.

Delta-based Hypergraph Construction

Since the updated incidence of modified vertices in δ`V2E` and modified hyperedges in δ`E2V` can be computed independently, we perform these computations concurrently by assigning them to different CUDA streams. A CUDA stream runs a sequence of GPU operations in first-in, first-out order. Using multiple streams allows independent operations to execute concurrently [73]. In the remaining sections, we focus on computing δ`V2E`, as the same method can be applied to compute δ`E2V`.

To compute δ`V2E`, we assign each modified vertex v to a GPU warp (32 threads). Each warp applies the modifiers involving v sequentially to update its original incidence in `V2E` and writes the result to δ`V2E`. Since both `V2E` and δ`V2E` reside in high-latency global memory, we use low-latency shared memory to avoid frequent global memory accesses. Specifically, we assign the warp a shared-memory segment whose capacity exceeds the size of v's original incidence to accommodate insertions, and initialize all entries in the segment to the empty entry $\varnothing$. The warp then loads the original incidence of v into its designated shared-memory segment, applies v's associated modifiers to update its incidence in shared memory, and writes the fully updated incidence back to δ`V2E`. This design significantly reduces memory latency, thereby improving kernel performance. To further optimize our kernel efficiency, we leverage CUDA warp-level primitives to efficiently handle deletion and insertion modifiers for each warp.

Hyperedge Deletion

For the warp to apply a deletion modifier $M^{-}_{(v,e)}$ to v's incidence in shared

Algorithm 14: Warp-level Deletion

```
Input  : $M^{-}_{(v,e)}$, $v$'s incident hyperedges in shared memory, $smem\_e$
1  parallel for each thread in a GPU warp
2      $lane\_id \leftarrow$ thread index in the GPU warp
3      $e\_cnt \leftarrow 0$
4      for j $\leftarrow 0$ to $SHARE\_SIZE/WARP\_SIZE - 1$ do
5          $entry \leftarrow smem\_e[j \times \texttt{WARP\_SIZE} + lane\_id]$
6          $all\_empty \leftarrow$ __all_sync(FULL, $entry == \varnothing$)
7          if $all\_empty > 0$
8              return
9          $active \leftarrow$ __ballot_sync(FULL, $entry \neq \varnothing \land entry \neq e$)
10         $rank \leftarrow$ __popc($active$&((1u<< $lane\_id$) $-$ 1u))
11         if $entry \neq \varnothing \land entry \neq e$
12             $smem\_e[e\_cnt + rank] \leftarrow entry$
13         if $lane\_id == 0$
14             $e\_cnt \leftarrow e\_cnt += $ __popc($active$)
```

memory, we invoke our warp-level deletion algorithm, where all threads cooperatively identify and remove the incident hyperedge e, and shift the remaining hyperedges to ensure that v's incidence remains contiguous using efficient CUDA warp primitives. Algorithm 14 presents our warp-level deletion algorithm. The warp iterates over all entries in shared memory, and in each iteration assigns one entry to each thread to process in parallel (lines 4-5). The threads then employ the CUDA warp-level primitive `__all_sync` to cooperatively check, via fast intra-warp communication, whether all threads' assigned entries are empty (line 6). If this condition is met, it indicates that all incident hyperedges of v have been processed, and the threads terminate early (lines 7-8). Otherwise, the threads cooperatively filter out those whose assigned entries are either empty or equal to the hyperedge e using `__ballot_sync` (line 9). The remaining threads then compute their compacted indices using `__popc`, and store the content of their assigned entries to the corresponding compacted locations in shared memory (lines 10-12). Finally, the first thread updates the current hyperedge count by adding the number of remaining threads (lines 13-14).

Hyperedge Insertion

For the warp to apply an insertion modifier $M^{+}_{(v,e)}$ to v's incidence in

Algorithm 15: Warp-level Insertion

```
   Input: $M^{+}(v, e)$, $v$'s incident hyperedges in shared memory, smem_e
1  parallel for each thread in a GPU warp
2      lane_id ← thread index in the GPU warp
3      for j ← 0 to SHARE_SIZE/WARP_SIZE − 1 do
4          entry ← smem_e[j × WARP_SIZE + lane_id]
5          if_empty ← __ballot_sync(FULL, entry == ∅)
6          first_empty_spot ← __ffs(if_empty) - 1
7          if (first_empty_spot ≠ −1) ∧ (lane_id == first_empty_spot)
8              smem_e[j × WARP_SIZE + lane_id] = e
9              break
```

shared memory, we invoke our warp-level insertion algorithm, where all threads cooperatively search for an empty entry in shared memory and replace it with e. Algorithm 15 presents our warp-level insertion algorithm. As in the deletion algorithm, the warp iterates over all entries in shared memory by assigning one entry per thread (lines 3-4). All the threads then cooperatively identify the first thread whose assigned entry is empty using `__ballot_sync` and `__ffs` (lines 5-6). If such a thread exists, that thread updates its assigned entry in shared memory with e and the algorithm terminates early (lines 7-9). Otherwise, all threads move on to the next iteration to continue looking for an empty entry.

4.5 Incremental Hypergraph Partitioning

Once the hypergraph is modified, the existing partitioning result may become invalid due to changes in the hypergraph structure and balance condition. For example, inserting or deleting vertices can violate the balance constraint, while modifying hyperedges' incident vertices may require moving them to reduce the cut size. To refine the modified hyperedge, a straightforward approach is to apply HyperG [67] to repartition it from scratch. However, this can incur significant overhead due to redundant computations, as most of the existing partitioning remains valid after small modifications. For instance, modifying the vertices incident to a hyperedge e may not affect the cut size if e already spans many partitions (i.e., has large $\lambda(e)$). Inserting or deleting a vertex from e's incidence may not change $\lambda(e)$, and thus has no impact on the overall cut size.

To address this problem, we propose an efficient IHP approach that quickly restores partition balance and refines only the vertices critical to the cut size, thereby avoiding redundant computations. Our approach consists of two steps: *single-pass rebalancing* and *incremental hypergraph refinement*, described below.

Single-pass Rebalancing

The goal of this step is to restore partition balance after the hypergraph has been modified. To do that, a common approach is to move vertices from overweight partitions to underweight ones until all partitions satisfy the balance constraint. However, concurrently moving vertices across

Algorithm 16: Single-pass Rebalancing

```
    Input: $V2E$, $V2EP$, $\delta V2E$, $\delta V2EP$
 1  Sort vertices by partition ID
 2  parallel for each thread in a GPU warp
 3      if $P(v)$ is not overweight
 4          return
 5      start ← v is modified ? $\delta V2EP[v]$ : $V2EP[v]$
 6      end ← v is modified ? $\delta V2EP[v+1]$ : $V2EP[v+1]$
 7      incidence ← v is modified ? $\delta V2E[start : end]$ : $V2E[start : end]$
 8      foreach $e \in$ incidence do
 9          Threads cooperatively check if $v$ is cut-critical to $e$ and update its score
10  Segmented sort vertices by descending score using ModernGPU
11  Move top-scoring vertices from each overweight partition to $P_{pseudo}$
```

partitions may require many iterations to achieve balance, especially when the number of partitions is large. For instance, if many vertices attempt to move to the same underweight partition, some moves must be rejected to prevent overloading that partition, and those vertices need to find other partitions in subsequent iterations. This iterative process largely underutilizes the massive parallelism available on a GPU.

To overcome this challenge, we introduce a *single-pass rebalancing* algorithm that reduces the weight of overweight partitions to achieve balance in just one iteration. Specifically, for each overweight partition, we move the minimal number of vertices required to reduce its weight below $W_{p_{max}}$ simultaneously to a *pseudo partition*. Since these vertices are not moved directly to underweight partitions, our approach avoids overloading any partition and allows them to be moved in parallel. Moreover, to prioritize moving vertices that may reduce the cut size, we assign a score to each vertex in the overweight partition and prioritize those with higher scores. The score of a vertex v is defined as $s(v) = c(v) - n(v)$, where $c(v)$ and $n(v)$ denote the number of incident hyperedges in which v is *cut-critical* and *non-cut-critical*, respectively. A vertex v is considered *cut-critical* to a hyperedge e if it is the only pin of e in its current partition, as moving v can reduce the connectivity of e and potentially reduce the cut size.

Otherwise, if v is not the only pin of e in its partition, it is considered *non-cut-critical*. A higher score indicates that v is cut-critical to most of its incident hyperedges and is more likely to reduce the cut size if moved.

Algorithm 16 presents our single-pass rebalancing algorithm. We first sort the vertices based on their partition IDs and assign each vertex v to a warp (lines 1-2). Each warp checks if v's partition is overweight. If not, the warp terminates early, as no rebalancing is needed for that partition (lines 3-4). Otherwise, the warp fetches its incident hyperedges from `δV2E` if v has been modified, or from `V2E` otherwise (lines 5-7). All threads in the warp then cooperatively process v's incident hyperedges one at a time to determine whether v is *cut-critical* to each hyperedge and update its score (lines 8-9). We then use ModernGPU [74]'s segmented sort to sort vertices by descending score within each overweight partition (line 10). Finally, we move the minimal number of top-scoring vertices needed to restore balance to a pseudo partition in parallel (line 11).

Incremental Hypergraph Refinement

The goal of this step is to identify vertices that require refinement due to hypergraph modifications. We then move these vertices to the pseudo partition and refine all the vertices in the pseudo partition in parallel. To do so, we first move newly inserted vertices to the pseudo partition, as they do not have any partition assignment and must be assigned one. In addition, since changes in the incidence of hyperedges can affect the cut size, we examine each modified hyperedge e and place each of its incident vertices v that is *cut-critical* to e into the pseudo partition, as moving v can reduce the connectivity of e by relocating it to a partition where e already has pins. Once the vertices are placed in the pseudo partition, we refine them in parallel by moving each vertex to its most suitable partition. The most suitable partition for a vertex v is the partition p_i to which v can be moved without violating the balance constraint and that

Algorithm 17: Most Suitable Partition Computation

```
Input: $V2E$, $V2EP$, $\delta V2E$, $\delta V2EP$
1  parallel for each vertex $v$ assigned to a GPU warp
2      $start \leftarrow v$ is modified? $\delta V2EP[v] : V2EP[v]$
3      $end \leftarrow v$ is modified? $\delta V2EP[v+1] : V2EP[v+1]$
4      $num\_e \leftarrow end - start$
5      $incidence \leftarrow v$ is modified? $\delta V2E[start : end] : V2E[start : end]$
6      $p_{ms} \leftarrow 0$
7      $\Delta\lambda_{v\rightarrow p_{ms}} \leftarrow$ INT_MAX
8      foreach $i \in [0, k-1]$ such that $W_{p_i} + W_v < W_{p_{max}}$ do
9          $\Delta\lambda_{v\rightarrow p_i} \leftarrow 0$
10         for $j \leftarrow 0$ to $\lceil num\_e/WARP\_SIZE \rceil - 1$ do
11             $e\_idx \leftarrow j \times$ WARP_SIZE $+ lane\_id$
12             $e \leftarrow e\_idx < num\_e \; ? \; incidence[e\_idx] : \varnothing$
13             $\Delta\lambda(e) \leftarrow$ (#pins of e in $p_i == 0$) ? 1 : 0
14             $sum \leftarrow$ __reduce_add_sync(FULL, $\Delta\lambda(e)$)
15             if $lane\_id == 0$
16                 $\Delta\lambda_{v\rightarrow p_i} += sum$
17         if $\Delta\lambda_{v\rightarrow p_i} < \Delta\lambda_{v\rightarrow p_{ms}} \wedge lane\_id == 0$
18             $\Delta\lambda_{v\rightarrow p_{ms}} \leftarrow \Delta\lambda_{v\rightarrow p_i}$
19             $p_{ms} \leftarrow p_i$
```

minimizes $\Delta\lambda_{v\rightarrow p_i}$. The term $\Delta\lambda_{v\rightarrow p_i}$ represents the number of v's incident hyperedges whose connectivity would increase if v were moved to p_i. By selecting the partition with the smallest increase in connectivity, we give cut-critical vertices a chance to reduce the cut size by moving them to a partition where most of their incident hyperedges already have pins.

However, moving vertices in parallel can result in the incorrect selection of the most suitable partition when adjacent vertices (i.e., vertices that share the same hyperedge) are moved simultaneously [67]. To address this issue, we select non-adjacent vertices from the pseudo partition and move them in parallel. Algorithm 17 presents our most suitable partition computation algorithm, where each non-adjacent vertex v is assigned to a GPU warp to compute its most suitable partition p_{ms} and the corresponding $\Delta\lambda_{v\rightarrow p_{ms}}$. The warp first fetches v's incidence information from $\delta V2E$ if v has been modified, or from $V2E$ otherwise (lines 1-5). Next, the warp iterates over each partition p_i that allows v to move without violating the balance constraint, and computes the corresponding $\Delta\lambda_{v \to p_i}$ (line 8). To compute $\Delta\lambda_{v \to p_i}$, each thread examines one of v's incident hyperedges e to check whether e has any pins in p_i. If not, the thread sets $\Delta\lambda(e) = 1$, indicating that moving v to p_i would increase the connectivity of e by one (lines 11-13). Threads then use `__reduce_add_sync` to efficiently compute the sum of all threads' $\Delta\lambda(e)$ values across the warp and increment $\Delta\lambda_{v \to p_i}$ accordingly (lines 14-16). All threads then repeat this procedure for the next 32 hyperedges, continuing until all of v's incident hyperedges have been examined. Finally, the first thread compares $\Delta\lambda_{v \to p_i}$ with $\Delta\lambda_{v \to p_{ms}}$, and updates the most suitable partition p_{ms} to p_i if it results in a smaller increase in connectivity (lines 17-19).

After computing the most suitable partition for each non-adjacent vertex, we sort them in ascending order of $\Delta\lambda_{v \to p_{ms}}$ to prioritize moving vertices with smaller increases in connectivity. We then adopt G-kway [18]'s sequence-based strategy to determine the maximum number of vertices that can be moved without violating the balance constraint and move all non-adjacent vertices in parallel. We repeat this process until all vertices in the pseudo partition have been moved to their most suitable partitions.

4.6 Experimental Evaluation

We evaluated the performance of iHyperG on 18 industrial circuit graphs derived from the ISPD98 VLSI Circuit Benchmark Suite [33]. Since the original circuits are small (a few thousand vertices), we expanded them 100–1000 times larger with random vertex and hyperedge insertions to demonstrate the advantage of GPU parallelism. In our experiment, we applied 100 incremental iterations based on the setting of TAU 2015 Incremental Timing Contest [64] and iG-kway [68], where each iteration involves tens to hundreds of hypergraph modifiers that randomly remove/insert vertices and hyperedges from/into the circuits.

We used HyperG [67], a state-of-the-art GPU-accelerated k-way hypergraph partitioner, as our baseline. Since HyperG does not support IHP, we rebuilt its hypergraph CSR on the GPU and repartitioned the modified circuit from that CSR. While this rebuilding strategy may not be optimal, it represented a straightforward extension of FHP to IHP. Further optimization of this is beyond the scope of this work.

To highlight the advantages of iHyperG's incremental approach in both hypergraph modification and partitioning, we report modification and partitioning time separately in the overall performance discussion. We implemented iHyperG and HyperG using C++17 and CUDA 12.0 and compiled them with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 16 Intel i7-11700 CPU cores at 2.50 GHz and 128 GB RAM. Our GPU was an A6000 with 48 GB of memory. All results are averaged over 10 runs.

Overall Performance Comparison

Table 4.1 compares the runtime and cut size between iHyperG and HyperG at $k = 2$ over 100 incremental iterations, each with 25 modifiers. A cut size improvement greater than one indicates that iHyperG achieves a smaller cut size than HyperG. To highlight the advantages of iHyperG, we break down the runtime into hypergraph modification and hypergraph partitioning. Since HyperG rebuilds the entire CSR at each iteration, iHyperG is consistently faster in the modification stage. Instead of rebuilding, iHyperG maintains a delta-based hypergraph data structure that records the updates to modified vertices and hyperedges. Moreover, to repartition from scratch, HyperG must rebuild an additional vertex–neighbor structure. In contrast, iHyperG avoids full repartitioning and therefore does not rebuild this structure, saving significant time in the modification stage. As a result, iHyperG achieves an average speedup of 190$\times$ and up to 402× over HyperG in modification time.

For partitioning time, iHyperG outperforms HyperG on all circuits with an average speedup of 83×. This speedup comes from iHyperG's incremental refinement, which efficiently identifies and refines only cut-critical vertices at each iteration. In contrast, HyperG repartitions the entire circuit every time, which quickly accumulates into substantial runtime overhead. In terms of cut size, iHyperG finds nearly the same cut size as HyperG (within ±2%). We attribute this to iHyperG's effective incremental refinement, which identifies cut-critical vertices within modified hyperedges and places them in their most suitable partitions to improve partitioning quality.

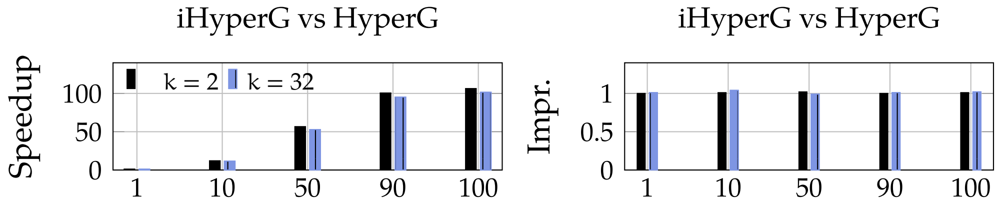


Figure 4.3: The speedup (left) and cut size improvement (right) of iHyperG over HyperG on Circuit05 over 100 incremental iterations, each with 25 modifiers.

Figure 4.3 shows the speedup of total partitioning time (modification and partitioning) and cut size improvement over 100 incremental iterations for Circuit05 at two extreme k values. At the first iteration, there is no significant difference in total runtime between iHyperG and HyperG since both are FHP. However, as the number of incremental iterations increases, iHyperG's runtime advantage over HyperG becomes more pronounced. The speedup of iHyperG grows roughly in proportion to the number of incremental iterations for both k values. This is because HyperG repartitions the circuit from scratch at each iteration, whereas iHyperG refines only the cut-critical vertices, saving substantial partitioning time per iteration. For cut size, iHyperG consistently matches HyperG across all iterations. This matching in partitioning quality demonstrates that, under small modifications, incremental partitioning achieves comparable results at a fraction of the runtime, making it a more efficient alternative to full repartitioning.

Runtime and Cut Size Analysis under Varying k

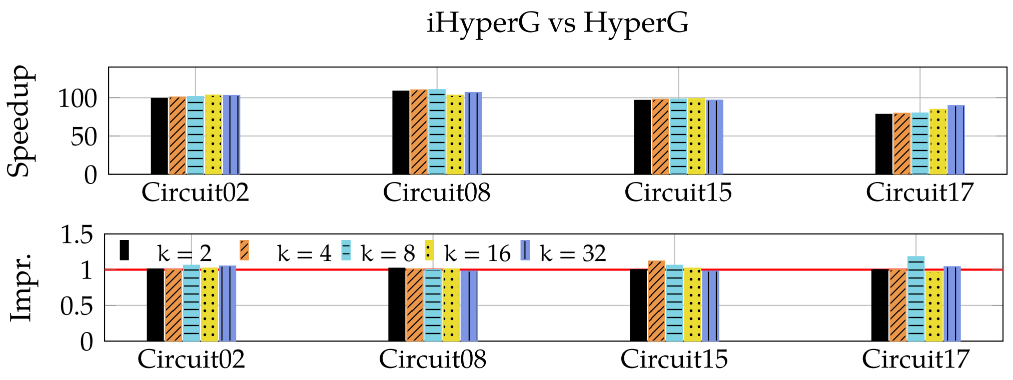


Figure 4.4: The speedup (top) and cut size improvement (bottom) of iHyperG over HyperG at different k values after 100 incremental iterations, each with 25 modifiers.

Figure 4.4 shows the speedup of total partitioning time (modification and partitioning) and cut size improvement of iHyperG over HyperG at $k = \{2, 4, 8, 16, 32\}$ on four circuits chosen to span a wide size range, including the smallest (Circuit08) and the largest (Circuit17). Across all k values, iHyperG achieves similar speedups on every circuit. As k increases, both iHyperG and HyperG must evaluate more partition assignments to refine vertices. Consequently, their runtimes grow similarly with k, and the speedup remains nearly constant.

For the cut size, iHyperG consistently matches HyperG and achieves better results on Circuit15 at $k = 4$ and Circuit17 at $k = 8$. We attribute this to iHyperG's incremental approach. With a small number of modifiers, the current partition provides a strong starting point, and incremental refinement preserves useful local connectivity, sometimes yielding a smaller cut size. In contrast, HyperG discards the current partition and repartitions everything from scratch, which can sometimes make it harder to find a high-quality partition that would otherwise be available through small and local refinement. These high-quality results demonstrate iHyperG's effectiveness and efficiency for IHP.

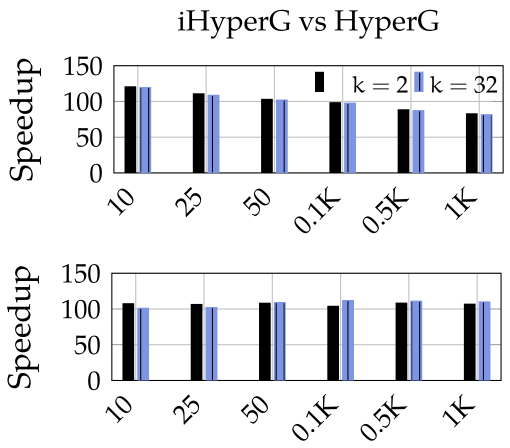


Figure 4.5: Speedup in hypergraph modification time (left) and hypergraph partitioning time (right) for iHyperG over HyperG on Circuit05 after 100 incremental iterations, with 10–1K modifiers per iteration.

Incrementality Analysis

Figure 4.5 shows the speedup in hypergraph modification and hypergraph partitioning time for iHyperG over HyperG on Circuit05 after 100 incremental iterations, with 10–1K modifiers per iteration. For hypergraph modification, the speedup is similar for both k values. However, it decreases as the number of modifiers grows because iHyperG must update larger delta structures with more modified vertices and hyperedges. On the other hand, HyperG rebuilds the CSR every time, so its modification time remains roughly constant even as the number of modifiers increases.

For hypergraph partitioning, we do not observe a strong dependence on the number of modifiers or the value of k. This is because hyperedges typically contain many pins, so a single pin change often does not affect whether the hyperedge is cut, leaving the overall cut size unchanged. Building on this insight, our incremental refinement refines only cut-critical vertices, keeping the refinement workload relatively stable even as the number of modifiers increases.

4.7 Conclusion

In this chapter, we present iHyperG, a GPU-parallel incremental k-way hypergraph partitioner. iHyperG introduces a scalable delta-based data structure for efficient hypergraph modifications on a GPU, along with an incremental partitioning algorithm that effectively refines the partitioning result. Experimental results show that iHyperG achieves average speedups of $190\times$ for modification and $83\times$ for partitioning over the state-of-the-art GPU-parallel full partitioner, HyperG [67], while maintaining comparable partitioning quality.

In this work, Wan Luan Lee was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and over- sight throughout the project. All authors contributed to the preparation and review of the final manuscript.

Table 4.1: Runtime (modification and partitioning) and cut size comparison between iHyperG and HyperG at $k = 2$ over 100 incremental iterations, each with 25 modifiers. All times are measured in seconds. A cut size improvement above one indicates that iHyperG finds a better cut size.

Benchmark			Hypergraph Modification Time (s)			Hypergraph Partitioning Time (s)			Cut Size		
Name	**# Vertices**	**# Edges**	**iHyperG**	**HyperG**	**Speedup**	**iHyperG**	**HyperG**	**Speedup**	**iHyperG**	**HyperG**	**Impr.**
Circuit01	2,639,746	2,921,135	0.07	3.71	53.00×	0.67	51.01	76.13×	1,526	**1510**	0.99
Circuit02	5,076,703	5,072,388	0.10	23.46	234.60×	2.03	187.26	92.25×	**1,572**	1,591	1.01
Circuit03	3,216,039	3,808,881	0.08	11.62	145.25×	1.16	92.54	79.78×	**1,680**	1,697	1.01
Circuit04	3,273,461	3,804,547	0.08	29.13	364.13×	1.20	98.01	81.68×	**1,699**	1,708	1.00
Circuit05	5,898,849	5,717,785	0.11	11.87	107.91×	2.83	300.45	106.17×	**841**	853	1.01
Circuit06	5,817,270	6,233,993	0.11	13.03	118.45×	1.70	141.10	83.00×	1,720	**1,701**	0.99
Circuit07	5,649,026	5,918,543	0.11	10.78	98.00×	1.62	133.25	82.25×	1,748	**1,747**	1.00
Circuit08	2,001,099	1,970,143	0.07	28.13	401.86×	1.26	116.04	92.10×	**853**	868	1.02
Circuit09	4,965,854	5,664,015	0.10	10.36	103.60×	1.18	94.79	80.33×	1,800	**1,795**	1.00
Circuit10	6,179,294	6,692,566	0.11	37.03	336.64×	1.74	138.53	79.61×	**1,821**	1,825	1.00
Circuit11	5,856,441	6,760,832	0.11	10.42	94.73×	1.41	103.53	73.43×	1,802	1802	1.00
Circuit12	3,767,136	4,285,768	0.09	23.78	264.22×	1.63	129.32	79.34×	**1,810**	1,811	1.00
Circuit13	3,620,692	4,285,768	0.09	8.82	98.00×	1.16	81.03	69.85×	1,837	**1,835**	1.00
Circuit14	5,166,292	5,347,138	0.10	21.72	217.20×	1.50	123.19	82.13×	1,849	**1,848**	1.00
Circuit15	7,917,046	9,143,924	0.14	47.22	337.29×	2.15	173.71	80.80×	883	883	1.00
Circuit16	7,889,952	8,172,196	0.13	19.11	147.00×	2.16	180.19	83.42×	1,866	1,866	1.00
Circuit17	11,686,333	11,943,736	0.17	23.39	137.59×	3.70	274.99	74.32×	**1,861**	1,870	1.00
Circuit18	7,371,509	7,067,343	0.12	19.64	163.67×	2.50	235.51	94.20×	1,852	1,852	1.00
Average					**190.17**×			**82.85**×			**1.00**

5 CONCLUSION

This dissertation addresses the growing demand for scalable and efficient graph and hypergraph partitioning algorithms in modern CAD workflows. As circuit sizes continue to increase, traditional CPU-based partitioners fall short of meeting the stringent runtime and scalability requirements of industrial-scale VLSI designs. In addition to full partitioning, incremental partitioning plays a critical role in many CAD algorithms, where it is integrated into iterative optimization workflows. Without incremental partitioning, the overhead of repetitive full partitioning can accumulate significantly, and the benefits of hypergraph partitioning cannot be fully exploited.

To overcome these limitations, this dissertation presents four complementary contributions. The first two are GPU-accelerated full partitioning algorithms: G-kway for graphs and HyperG for hypergraphs. Both leverage the massive parallelism of a GPU to accelerate the partitioning beyond what is achievable with traditional CPU-parallel approaches. The last two are GPU-parallel incremental partitioning algorithms: iG-kway for graphs and iHyperG for hypergraphs. Both design incrementality-aware data structures that enable efficient graph and hypergraph modifications on the GPU and selectively refine only the vertices requiring refinement. This approach avoids the substantial overhead of repeatedly repartitioning the entire graph or hypergraph during iterative optimization workflows.

In summary, this dissertation advances the state-of-the-art in scalable and efficient partitioning algorithms for graph and hypergraph workloads in modern CAD workflows. It demonstrates how GPU parallelism, combined with incrementality-aware data structures and refinement strategies, can be effectively leveraged to develop high-performance partitioners that meet the stringent runtime and scalability demands of industrial-scale VLSI systems.

Inspired by a series of GPU-related research developed in our research group [75, 76, 77, 68, 70, 24, 78, 79, 80, 67, 81, 82, 83, 84, 85, 86, 87, 88, 18, 89, 71, 90, 91, 92, 93, 94, 95, 96, 97, 98, 44, 99, 100, 53, 54, 101, 102, 103, 12, 104, 105, 106, 107, 22, 108, 109, 110, 55, 111, 112, 23, 56, 113, 114, 11, 115, 116, 117, 118, 119, 21, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133], future work may extend the proposed frameworks to support distributed multi-GPU platforms and multi-constraint partitioning, which would further improve scalability and applicability to large design scenarios. Another promising direction is the integration of machine learning techniques to guide coarsening and refinement based on the structural characteristics of graphs and hypergraphs. These enhancements could enable more adaptive and intelligent partitioning workflows.

BIBLIOGRAPHY

[1] D. LaSalle and G. Karypis, “Multi-threaded graph partitioning,” in *IPDPS*. IEEE, 2013.

[2] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag, “Scalable shared-memory hypergraph partitioning,” in *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2021, pp. 16–30.

[3] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*. Springer, 2011, vol. 312.

[4] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[5] U. V. Catalyurek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

[6] T. N. Bui and C. Jones, “Finding good approximate vertex and edge partitions is np-hard,” *Information Processing Letters*, vol. 42, no. 3, pp. 153–159, 1992.

[7] I. Bustany, G. Gasparyan, A. B. Kahng, I. Koutis, B. Pramanik, and Z. Wang, “An open-source constraints-driven general partitioning multi-tool for vlsi physical design,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[8] U. Çatalyürek, K. Devine, M. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner, “More Recent Advances in (Hyper)Graph Partitioning,” *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023.

[9] R. Liang, A. Agnesina, and H. Ren, "Medpart: A multi-level evolutionary differentiable hypergraph partitioner," in *Proceedings of the 2024 International Symposium on Physical Design*, 2024, pp. 3–11.

[10] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," vol. 20, no. 1. SIAM, 1998.

[11] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

[12] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," in *ACM International Conference on Parallel Processing (ICPP)*, 2022.

[13] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *ACM/IEEE DAC*, 1982, pp. 175–181.

[14] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in vlsi domain," in *DAC*, 1997.

[15] Y. Liu, R. Ye, F. Yuan, R. Kumar, and Q. Xu, "On logic synthesis for timing speculation," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 591–596.

[16] B. Goodarzi, M. Burtscher, and D. Goswami, "Parallel graph partitioning on a cpu-gpu architecture," in *IPDPSW*. IEEE, 2016.

[17] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami, "High performance multilevel graph partitioning on gpu," in *HPCS*. IEEE, 2019.

[18] W. L. Lee, D.-L. Lin, T.-W. Huang, S. Jiang, T.-Y. Ho, Y. Lin, and B. Yu, "G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[19] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *PACT*. IEEE, 2015.

[20] NVIDIA, "Cuda graphs: Accelerating your gpu workloads," https://developer.nvidia.com/blog/cuda-graphs/, 2019, accessed: 2024-08-23.

[21] D.-L. Lin and T.-W. Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2020.

[22] ——, "Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[23] ——, "Efficient GPU Computation using Task Graph Parallelism," in *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2021.

[24] S. Jiang, Y.-H. Chung, C.-C. Chang, T.-Y. Ho, and T.-W. Huang, "BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[25] T.-W. Huang and M. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[26] S. Maleki, U. Agarwal, M. Burtscher, and K. Pingali, "Bipart: a parallel and deterministic hypergraph partitioner," in *Proceedings*

of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021, pp. 161–174.

[27] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[28] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 3558–3565.

[29] E. Ihler, D. Wagner, and F. Wagner, "Modeling hypergraphs by graphs with the same mincut properties," *Information Processing Letters*, vol. 45, no. 4, pp. 171–175, 1993.

[30] L. Cheng, H. Cho, and P. Yoon, "An accelerated procedure for hypergraph coarsening on the gpu," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–7.

[31] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag, "Engineering a direct k-way hypergraph partitioning algorithm," in *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2017, pp. 28–42.

[32] K. Yonehara and K. Aizawa, "A line-based connected component labeling algorithm using gpus," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 341–345.

[33] C. J. Alpert, "The ispd98 circuit benchmark suite," in *Proceedings of the 1998 international symposium on Physical design*, 1998, pp. 80–85.

[34] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam, "Jet: Multilevel graph partitioning on graphics processing units," *SIAM Journal on Scientific Computing*, vol. 46, no. 5, pp. B700–B724, 2024.

[35] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, "Parallel graph partitioning on multicore architectures," in *LCPC*. Springer, 2011.

[36] D. LaSalle and G. Karypis, "A parallel hill-climbing refinement algorithm for graph partitioning," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 236–241.

[37] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2710–2722, 2020.

[38] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, "Deep multilevel graph partitioning," *arXiv preprint arXiv:2105.02022*, 2021.

[39] S. V. Patil and D. B. Kulkarni, "Graph partitioning using heuristic kernighan-lin algorithm for parallel computing," in *Next Generation Information Processing System: Proceedings of ICCET 2020, Volume 2*. Springer, 2021, pp. 281–288.

[40] P. Sanders and D. Seemaier, "Distributed deep multilevel graph partitioning," in *European conference on parallel processing*. Springer, 2023, pp. 443–457.

[41] M. F. Faraj and C. Schulz, "Recursive multi-section on the fly: Shared-memory streaming algorithms for hierarchical graph partitioning and process mapping," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 473–483.

[42] P. Sanders and D. Seemaier, "Brief announcement: Distributed unconstrained local search for multilevel graph partitioning," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY, USA: Association

for Computing Machinery, 2024, p. 443–445. [Online]. Available: https://doi.org/10.1145/3626183.3660257

[43] B. Goodarzi, M. Burtscher, and D. Goswami, "Parallel graph partitioning on a cpu-gpu architecture," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 58–66.

[44] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *ACM/IEEE Design Automation Conference (DAC)*, 2023.

[45] L. Durbeck and P. Athanas, "Incremental streaming graph partitioning," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–8.

[46] W. Ju, J. Li, W. Yu, and R. Zhang, "igraph: an incremental data processing system for dynamic graph," *Frontiers of Computer Science*, vol. 10, pp. 462–476, 2016.

[47] W. Fan, M. Liu, C. Tian, R. Xu, and J. Zhou, "Incrementalization of graph partitioning algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1261–1274, 2020.

[48] D. Dai, W. Zhang, and Y. Chen, "Iogp: An incremental online graph partitioning algorithm for distributed graph databases," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 219–230.

[49] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," *IEEE transactions on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 884–896, 1997.

[50] S. Lin, J. Liu, E. F. Young, and M. D. Wong, "Gamer: Gpu-accelerated maze routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 583–593, 2022.

[51] S. Liu, Y. Pu, P. Liao, H. Wu, R. Zhang, Z. Chen, W. Lv, Y. Lin, and B. Yu, "Fastgr: Global routing on cpu–gpu with heterogeneous task graph scheduler," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 7, pp. 2317–2330, 2022.

[52] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020.

[53] G. Guo, T.-W. Huang, and M. D. F. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2023.

[54] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. Wong, "A GPU-Accelerated Framework for Path-Based Timing Analysis," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[55] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[56] ——, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[57] T. Liu, L. Chen, X. Li, M. Yuan, and E. F. Young, "Finemap: A fine-grained gpu-parallel lut mapping engine," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 392–397.

[58] E. F. Young, "Gpu acceleration in physical synthesis," in *Proceedings of the 2023 International Symposium on Physical Design*, ser. ISPD '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 167. [Online]. Available: https://doi.org/10.1145/3569052.3578912

[59] J. Jiang, L. Zou, W. Zhao, Z. He, T. Chen, and B. Yu, "Pdrc: Package design rule checking via gpu-accelerated geometric intersection algorithms for non-manhattan geometry," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3657367

[60] P. Liao, Y. Zhao, D. Guo, Y. Lin, and B. Yu, "Analytical die-to-die 3-d placement with bistratal wirelength model and gpu acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 6, pp. 1624–1637, 2024.

[61] Z. He, Y. Zuo, J. Jiang, H. Zheng, Y. Ma, and B. Yu, "Opendrc: An efficient open-source design rule checking engine with hierarchical gpu acceleration," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[62] Z. Yu, G. Chen, Y. Ma, and B. Yu, "A gpu-enabled level-set method for mask optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 594–605, 2023.

[63] Z. He, Y. Ma, and B. Yu, "X-check: Gpu-accelerated design rule checking via parallel sweepline algorithms," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549383

[64] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *IEEE/ACM ICCAD*, 2015, pp. 882–889.

[65] D. Guide, "Cuda c++ programming guide," *NVIDIA, July*, 2020.

[66] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th dimacs implementation challenge workshop," 2012.

[67] W.-L. Lee, D.-L. Lin, C.-H. Chiu, U. Schlichtmann, and T.-W. Huang, "HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[68] W.-L. Lee, S. Jiang, D.-L. Lin, C. Chang, B. Zhang, Y.-H. Chung, U. Schlichtmann, T.-Y. Ho, , and T.-W. Huang, "iG-kway: Incremental k-way Graph Partitioning on GPU," in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[69] W. L. Lee, D.-L. Lin, S. Jiang, C.-H. Chiu, Y. Lin, B. Yu, T.-Y. Ho, and T.-W. Huang, "G-kway: Multilevel gpu-accelerated k-way graph partitioner using task graph parallelism," *ACM Transactions on Design Automation of Electronic Systems*, 2025.

[70] Y.-H. Chung, S. Jiang, W. L. Lee, Y. Zhang, H. Ren, T.-Y. Ho, and T.-W. Huang, "SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[71] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W. L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, "G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis," in *ACM/IEEE DAC*, 2024.

[72] Z. Wu, H. Zhao, H. Liu, W. Wen, and J. Li, "ghypart: Gpu-friendly end-to-end hypergraph partitioner," *ACM Transactions on Architecture and Code Optimization*, vol. 22, no. 1, pp. 1–25, 2025.

[73] D. Guide, "Cuda c programming guide," *NVIDIA, July*, vol. 29, no. 31, p. 6, 2013.

[74] S. Baxter, "moderngpu 2.0," 2016, https://github.com/moderngpu/moderngpu/wiki.

[75] A. D. Sarma, S. Jiang, W.-L. Lee, T.-Y. Ho, and T.-W. Huang, "TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2026.

[76] J. Tong, W.-L. Lee, U. Y. Ogras, and T.-W. Huang, "Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[77] C.-C. Chang and T.-W. Huang, "Statistical Timing Graph Scheduling Algorithm for GPU Computation," in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[78] S. Gener, S. Hassan, L. Chang, C. Chakrabarti, T.-W. Huang, U. Ograss, , and A. Akoglu, "A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems," in *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*, 2025.

[79] C. Chang, B. Zhang, C.-H. Chiu, D.-L. Lin, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang, "PathGen: An Efficient Parallel Critical Path Generation Algorithm," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[80] B. Zhang, C. Chang, C.-H. Chiu, D.-L. Lin, Y. Sui, C.-C. Chang, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang, "iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[81] C.-H. Chiu, C. Morchdi, Y. Zhou, B. Zhang, C. Chang, and T.-W. Huang, "Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2024.

[82] C.-C. Chang, B. Zhang, and T.-W. Huang, "GSAP: A GPU-Accelerated Stochastic Graph Partitioner," in *ACM ICPP*, 2024, p. 565–575.

[83] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, "HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2024.

[84] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei, "FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array," in *ACM ICPP*, 2024, p. 388–399.

[85] J. Tong, L. Chang, U. Y. Ogras, and T.-W. Huang, "BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[86] C. Chang, C.-H. Chiu, B. Zhang, and T.-W. Huang, "Incremental Critical Path Generation for Dynamic Graphs," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[87] C.-H. Chiu and T.-W. Huang, "An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads," in *IEEE Computer Society Annual Symposium on VLSI* (*ISVLSI*), 2024.

[88] D.-L. Lin, T.-W. Huang, J. S. Miguel, and U. Ogras, "TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling," in *International European Conference on Parallel and Distributed Computing* (*Euro-Par*), 2024.

[89] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin, "Ink: Efficient Incremental k-Critical Path Generation," in *ACM/IEEE DAC*, 2024.

[90] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis," in *ACM/IEEE DAC*, 2024.

[91] C. Morchdi, C.-H. Chiu, W.-L. Lee, T.-W. Huang, and Y. Zhou, "Enhancing Graph Partitioning with Reinforcement Learning-based Initialization," in *IEEE High-performance and Extreme Computing Conference* (*HPEC*), 2025.

[92] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu, "Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System," in *ACM International Symposium on Physical Design* (*ISPD*), 2024.

[93] Z. Guo, T.-W. Huang, J. Zhou, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous Static Timing Analysis with Advanced Delay Calculator," in *IEEE/ACM Design, Automation and Test in Europe Conference* (*DATE*), 2024.

[94] C. Morchdi, C.-H. Chiu, Y. Zhou, and T.-W. Huang, "A Resource-efficient Task Scheduling System using Reinforcement Learning," in *IEEE/ACM Asia and South Pacific Design Automation Conference* (*ASP-DAC*), 2024.

[95] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2023.

[96] C.-C. Chang and T.-W. Huang, "uSAP: An Ultra-Fast Stochastic Graph Partitioner," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[97] S. Jiang, T.-W. Huang, and T.-Y. Ho, "GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[98] ——, "SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU," in *ACM International Conference on Parallel Processing (ICPP)*, 2023.

[99] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[100] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*, 2023.

[101] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[102] T.-W. Huang and L. Hwang, "Task-parallel Programming with Constrained Parallelism," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[103] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[104] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism using Control Taskflow Graph," in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[105] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms," in *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[106] T.-W. Huang and Y. Lin, "Concurrent CPU-GPU Task Programming using Modern C++," in *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*, 2022.

[107] K. Zhou, Z. Guo, T.-W. Huang, and Y. Lin, "Efficient Critical Paths Search Algorithm using Mergeable Heap," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022.

[108] M. Mower, L. Majors, and T.-W. Huang, "Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs," in *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2021.

[109] T.-W. Huang, "TFProf: Profiling Large Taskflow Programs with Modern D3 and C++," in *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*, 2021.

[110] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[111] Y. Zamani and T.-W. Huang, "A High-Performance Heterogeneous Critical Path Analysis Framework," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2021.

[112] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*, 2021.

[113] Z. Guo, T.-W. Huang, and Y. Lin, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[114] K.-M. Lai, T.-W. Huang, P.-Y. Lee, and T.-Y. Ho, "ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

[115] T.-W. Huang, C.-X. Lin, and M. Wong, "OpenTimer v2: A Parallel Incremental Timing Analysis Engine," *IEEE Design and Test (DAT)*, 2021.

[116] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[117] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE*

International Parallel and Distributed Processing Symposium (*IPDPS*), 2019.

[118] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (*TCAD*), 2022.

[119] C.-X. Lin, T.-W. Huang, and M. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE International Conference on Parallel and Distributed Systems* (*ICPADS*), 2020.

[120] T.-W. Huang, "A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM International Conference on Computer-aided Design* (*ICCAD*), 2020.

[121] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated Static Timing Analysis," in *IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD*), 2020.

[122] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE Design Automation Conference* (*DAC*), 2020.

[123] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM Multimedia Conference* (*MM*), 2019.

[124] ——, "An Efficient and Composable Parallel Task Programming Library," in *IEEE High-performance and Extreme Computing Conference* (*HPEC*), 2019.

[125] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A General Cache Framework for Efficient Generation of Timing Critical Paths," in *ACM/IEEE Design Automation Conference* (*DAC*), 2019.

[126] T.-W. Huang, C.-X. Lin, , and M. Wong, "Distributed Timing Analysis at Scale," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[127] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Essential Building Blocks for Creating an Open-source EDA Project," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[128] T.-W. Huang, C.-X. Lin, and M. Wong, "DtCraft: A High-performance Distributed Execution Engine at Scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[129] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "A General-purpose Distributed Programming System using Data-parallel Streams," in *ACM Multimedia Conference (MM)*, 2018.

[130] C.-X. Lin, T.-W. Huang, T. Yu, and M. Wong, "A Distributed Power Grid Analysis Framework from Sequential Stream Graph," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.

[131] T.-W. Huang, C.-X. Lin, and M. Wong, "DtCraft: A Distributed Execution Engine for Compute-intensive Applications," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2017.

[132] T.-Y. Lai, T.-W. Huang, , and M. Wong, "Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2017.

[133] T.-W. Huang, M. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A Distributed Timing Analysis Framework for Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2016.