# G-kway: Multilevel GPU-Accelerated $k$-way Graph Partitioner using Task Graph Parallelism

WAN LUAN LEE, University of Wisconsin-Madison, Madison, United States
DIAN-LUN LIN, University of Wisconsin-Madison, Madison, United States
SHUI JIANG, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong
CHENG-HSIANG CHIU, University of Wisconsin-Madison, Madison, United States
YIBO LIN, Peking University, Beijing, China
BEI YU, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong
TSUNG-YI HO, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong
TSUNG-WEI HUANG, University of Wisconsin-Madison, Madison, United States

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× and 3.8×, respectively, 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.

CCS Concepts: • **Hardware → Partitioning and floorplanning**.

## 1 Introduction

Graph partitioning is important for the design of efficient computer-aided design (CAD) algorithms [4, 19, 26, 30, 32, 40, 41] because it allows an algorithm to break down a problem into smaller and manageable pieces. Since graph partitioning is NP-complete [7, 17], prior research has proposed several frameworks and heuristics, including spectrum method [6, 37], FM-based refinement [5], geometric partitioning [16, 38], learning-based partitioning [3, 35], and multilevel graph partitioning [20]. Among various partitioning frameworks, *multilevel partitioning* is the most popular for large-scale graphs due to its high partitioning quality and fast runtime. [2, 34].

Authors' Contact Information: Wan Luan Lee, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: wanluan.lee@wisc.edu; Dian-Lun Lin, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: dianlun.lin@wisc.edu; Shui Jiang, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong; e-mail: sjiang22@cse.cuhk.edu.hk; Cheng-Hsiang Chiu, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: chenghsiang.chiu@wisc.edu; Yibo Lin, Peking University, Beijing, China; e-mail: yibolin@pku.edu.cn; Bei Yu, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong; e-mail: byu@cse.cuhk.edu.hk; Tsung-Yi Ho, The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong; e-mail: tyho@cse.cuhk.edu.hk; Tsung-Wei Huang, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: tsung-wei.huang@wisc.edu.


ACM 1557-7309/2025/5-ART
https://doi.org/10.1145/3734522

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. For example, partitioning a timing graph can take up to 60% of the runtime per timing update in a static timing analysis (STA) engine [11, 12, 15], and partitioning can be performed iteratively during incremental timing. Similar problems exist in many RTL simulation workloads as well [32]. To alleviate the long runtime, existing partitioners have leveraged multicore CPUs [1, 10, 22] to parallelize the partitioning algorithm. Despite some runtime improvements, the speedup is typically limited to only 8–16 CPU threads [22]. 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, [8] 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 [9] performs the entire graph partitioning on 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., eight) 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 [9]. To overcome these problems, we propose G-kway [25], 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 [22] and GPU-based GKSG [9]. 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.

## 2 Problem Definition and Notation

Given the importance of graph partitioning in designing efficient CAD algorithms, this section provides a formal description of the partitioning problem. 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 $e = (u, v)$, representing an edge connecting vertices $u$ and $v$ in $V$. We denote the neighbors of $v$ as $adj(v)$. 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

$$W_{p_i} \leq (1+\epsilon)\frac{\sum_{v\in V} W_v}{k}$$

, where $0 < \epsilon \ll 1$ and $\epsilon$ is the imbalance ratio given by applications.

In this paper, we focus on partitioning large regular graphs, which is widely applied in many CAD algorithms and beyond. For instance, Register-transfer level (RTL) simulators [31] partition a big logic graph into $k$ macro tasks that produce minimal cut dependencies for improving simulation performance; STA engine [15] partitions a million-task timing graph into $k$ min-cut subgraphs for reducing scheduling overhead.

## 3 GPU Multilevel $k$-way Partitioner

In this section, we present our multilevel GPU-based k-way graph partitioning method, G-kway. Figure 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 the coarsening threshold, $\gamma$ (typically $160 \times k$). 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 [20] 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 to its previous state and reduce the cut size of the restored graph by moving each vertex to a different 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. When the refinement algorithm has substantial kernel launch overhead, we use CUDA Graph to improve performance by reducing this overhead.

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. Moreover, for graphs requiring a significantly large number of iterative control flow operations (e.g., 2,000), we further enhance performance by utilizing CUDA Graph and a conditional node, allowing the GPU to manage the control flow directly and eliminating frequent CPU-GPU data transfers.

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 [9] for efficient GPU computing.

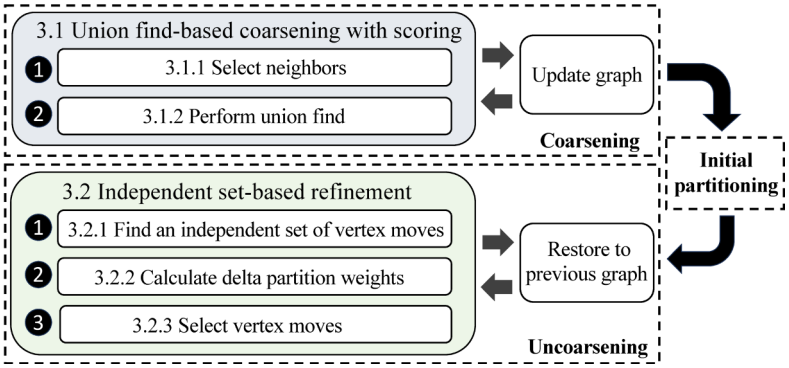


Fig. 1. Overview of G-kway that consists of three main stages: coarsening, initial partition, and uncoarsening.

## 3.1 Union Find-based Coarsening with Scoring

Most existing parallel multilevel graph partitioners such as GKSG [9] 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 2 shows the comparison between GKSG's coarsening algorithm and ours. As shown in Figure 2 (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.

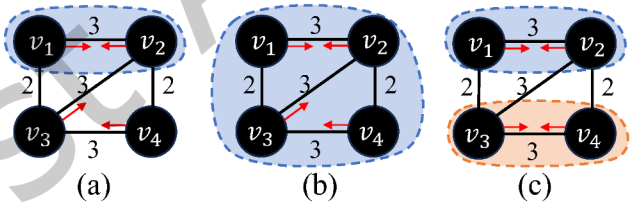


Fig. 2. Examples of three coarsening methods for one iteration, including (a) Heavy Edge Matching (HEM) by GKSG, (b) Union find-based coarsening without scoring, and (c) Union find-based coarsening with scoring. Each vertex has a red arrow pointing to its selected neighbor. Vertices circled in the same color are coarsened into a coarsened vertex.

To address these issues, our initial solution is to group 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 group the vertex into the same subset. This union find-based strategy eliminates the need for iteratively searching neighbors to form match pairs, ensuring each vertex can be grouped into a subset in a single iteration. Also, since we group 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 grouped into the same subset. As shown in 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 join a subset in one iteration, all vertices eventually join the same subset and are coarsened into a single vertex.

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 2 (c) shows our union find-based coarsening with scoring. $v_3$ selects $v_4$ 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.*

**Algorithm 1** Union find-based coarsening with scoring

```
1:  /* select neighbors: assign 32 vertices and their edges to a GPU warp */
2:  parallel for each thread in a warp {
3:      while (there are more edges to process) {
4:          get an edge e_i = (u, v) to process
5:          find the assigned edge's source vertex u
6:          s(u, v) ← c × W_(u,v) - degree(v)
7:          /* using __shfl_up_sync */
8:          reduce on the scores with threads have the same source vertex
9:      }
10:     write a vertex u's selected neighbor to selected_nbr array
11: }
12: /* union find: assign each vertex v_i to a GPU thread T_i */
13: while (any threads is still updating) {
14:     parallel for each thread in a warp {
15:         nbr ← selected_nbr[v_i]
16:         v_i_subset_ID ← d_subset_ID[v_i]
17:         nbr_subset_ID ← d_subset_ID[nbr]
18:         if (v_i_subset_ID > nbr_subset_ID) then
19:             atomicMin(&d_subset_ID[v_i], nbr_subset_ID)
20:         else if (v_i_subset_ID < nbr_subset_ID) then
21:             atomicMin(&d_subset_ID[nbr], v_i_subset_ID)
22:         /* using __any_sync */
23:         check if any thread in a warp updates subset_IDs
24:     }
25: }
```

*3.1.1 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 [21]. 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 [39]. 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).

*3.1.2 Perform union find.* After selecting the highest score neighbor for each vertex, we perform `union_find` to group vertices into subsets. 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 *nbr* 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 group 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.

## 3.2 Independent Set-based Refinement

The goal of the refinement algorithm is to reduce the cut size by moving a vertex to a partition that maximizes the reduction in cut size. 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 $u \notin p_i$. The internal degree of $u$, denoted as $id(u)$, is the sum of the weights of edges $(u, v)$ where both $u$ and $v$ belong to the same partition:

$$id(u) = \sum_{v \in \text{adj}(u), P(v)=P(u)} w(u, v)$$

. On the other hand, the external degree of $u$ to partition $p_i$, denoted as $ed(u, p_i)$, is the sum of the weights of edges $(u, v)$ where $v$ belongs to partition $p_i$:

$$ed(u, p_i) = \sum_{v \in \text{adj}(u), P(u)=i} w(u, v)$$

. 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 [9]. 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.

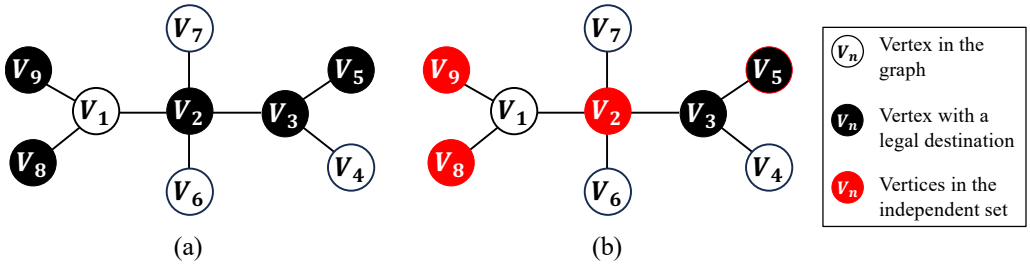


Fig. 3. Example of finding an independent set of vertices. Vertices colored black have a legal destination partition, while those colored red are added to the independent set. Each thread is assigned a vertex. In step (a), each thread checks whether its assigned vertex has a legal destination partition and marks it black if one exists. In step (b) each thread checks whether its assigned vertex is independent and marks it red if the conditions are met.

*3.2.1 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.

Figure 3 shows an example of finding an independent set of vertices. Vertices with a legal destination partition are shown in black, while vertices in the independent set are colored red. In this example, the thread assigned vertex $v_9$ adds it to the independent set because none of $v_9$'s neighbors have a legal destination partition. Additionally, the thread assigned to vertex $v_2$ first checks if $v_2$ has neighbors with a legal destination partition. Since $v_2$ has a neighbor with a legal partition, $v_3$, the thread compares their vertex IDs and adds $v_2$ to the move buffer because its vertex ID is smaller than that of $v_3$.

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 ← find a legal destination partition with the largest gain
8:          if (dst exists) then
9:              nbors ← 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: }
```

*3.2.2 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 1,024. Since most modern GPUs have 1,024 threads per GPU block, calculating more than 1,024 vertex moves needs multiple blocks for each segment, which requires expensive synchronization across multiple blocks.

Figure 4 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.

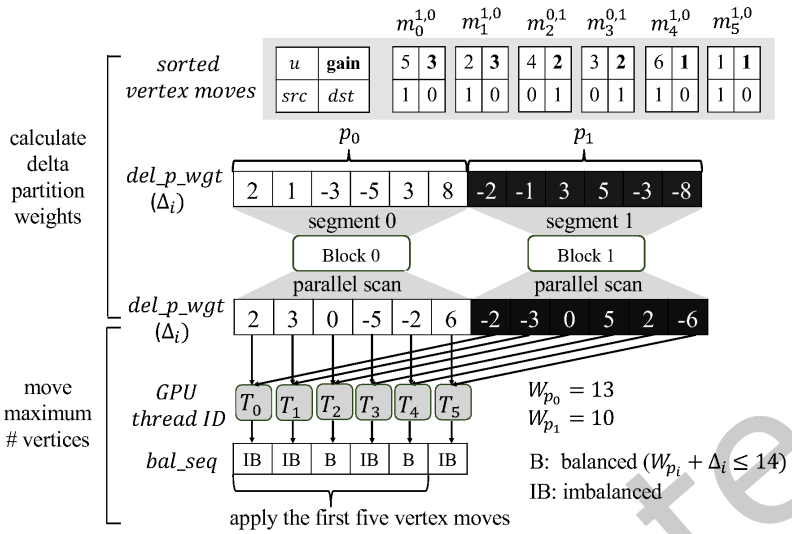


Fig. 4. Illustration of the process to construct *del_p_wgt* and *bal_seq* with $k = 2$ under six vertex moves. Assuming current partition weights are 13 and 10 for $p_0$ and $p_1$, respectively, with a maximum partition weight of 14.



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

**Algorithm 3** Calculate delta partition weights

```
1:  parallel sort the move buffer in descending order by gain
2:  seg_size ← min(#vertex_moves, 1024)
3:  gid ← 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 ← m_u^src,dst.src × seg_size + gid
7:      dst_p_idx ← m_u^src,dst.dst × seg_size + gid
8:      del_p_wgt[src_p_idx] ← -W_u
9:      del_p_wgt[dst_p_idx] ← 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 ← b_i.ID × seg_size
15:     seg_i_end ← seg_i_start + seg_size
16:     parallel scan on seg_i          /* __shfl_up_sync */
17:     return
18: }
```

*3.2.3 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 4, 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 $j^{th}$ results in a balanced partition. We then select the longest sub-sequence of vertex moves by finding the largest 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 4, 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 $W_{p0} + \Delta_0 \leq 14$ and $W_{p1} + \Delta_6 \leq 14$. 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 ($bal\_seq[0] = IB$). However, applying the first five vertex moves helps to restore the partition result back to balance ($bal\_seq[4] = B$). In the example shown in Figure 4, the longest sub-sequence is from the first to the fifth vertex moves. Because the sequence of vertex moves is sorted in descending order of gain, this allows us to prioritize 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.

### 3.3 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 5 illustrates the time spent on kernel execution with and without a CUDA Graph. Without CUDA Graphs, the host need to launch each kernel individually and each kernel launch incurs overhead. When the number of iteration 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.

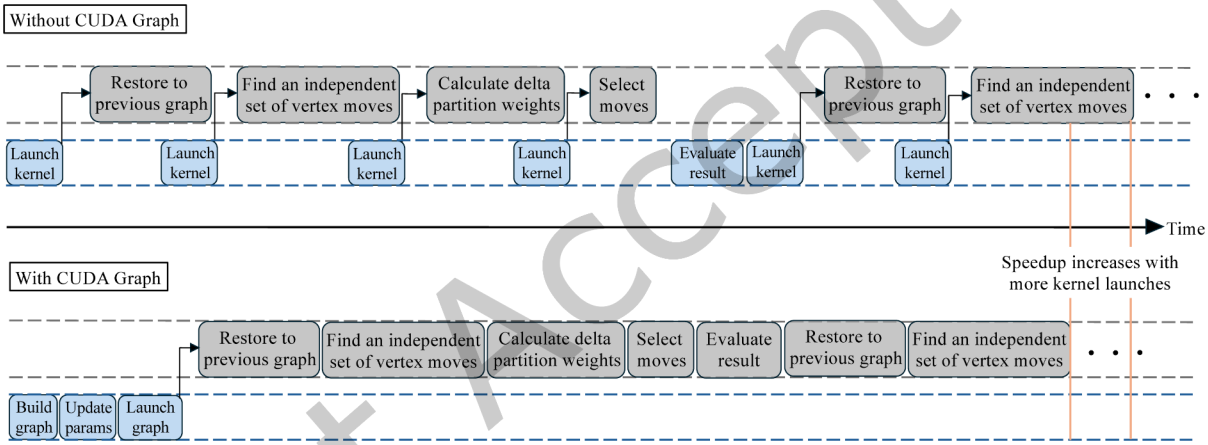


Fig. 5. The time spent on kernel execution with and without CUDA Graphs. Each gray box represents a GPU task, while the blue box indicates a CPU operation. The top graph shows the traditional approach without using CUDA Graphs, where each kernel launch incurs significant overhead. The bottom graph shows 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.

*3.3.1 CUDA Graph-based Execution Model.* 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 [36]. 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 [18, 27–29]. Fig 6 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 small cost.

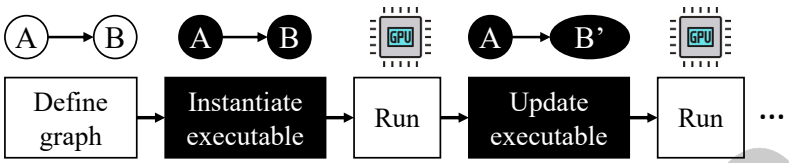


Fig. 6. Execution model of CUDA Graph consists of four major steps, graph definition, executable graph instantiation, graph launch (run), and graph parameter updates.



*3.3.2 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 kernel 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 7 illustrates the workflow of a sequence of GPU tasks involving a *while* dynamic control flow, without and with a conditional node. In Figure 7 (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 7 (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.

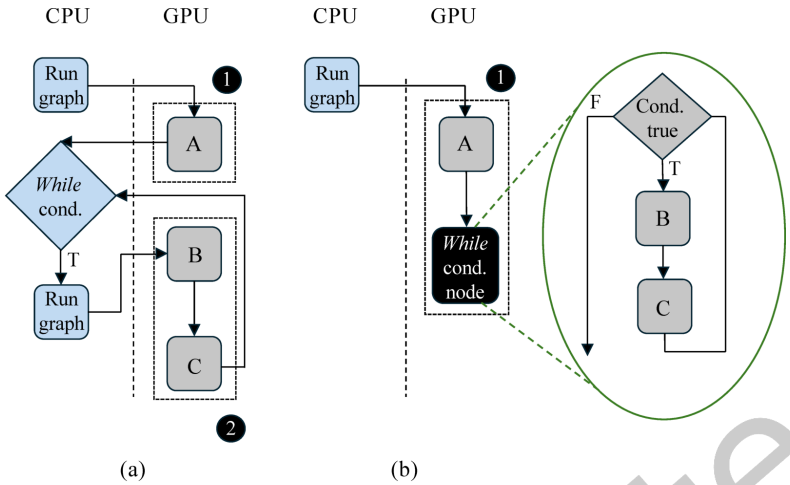


Fig. 7. An example of the workflow of a sequence of GPU tasks involving a *while* dynamic control flow. In Figure (a), the workflow is shown without a CUDA conditional node, and in Figure (b), the workflow is shown with a CUDA conditional node. Each gray box represents a graph node that contains a GPU 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.

*3.3.3 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 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 uncoarsneing 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 8 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* (see Section 3.2.1), *calculate delta partition weights* (see Section 3.2.2), and *select vertex moves* (see Section 3.2.3).

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 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 [36] 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

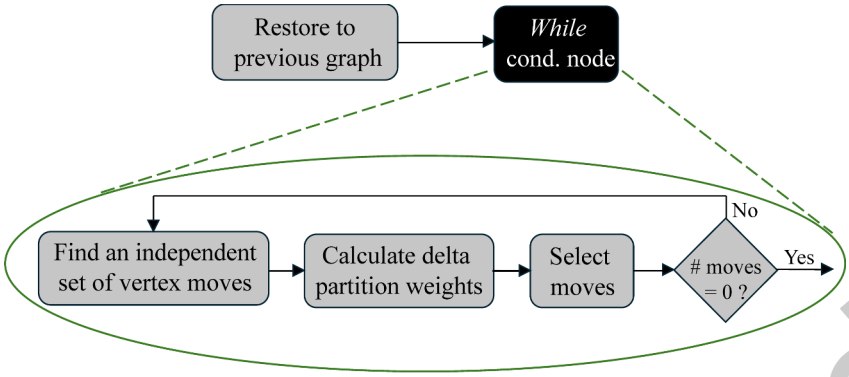


Fig. 8. The CUDA Graph encapsulates all GPU kernels used for uncoarsening. Within the graph, a CUDA conditional node iteratively evaluates the number of vertex moves to manage dynamic control flow on the GPU.

**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
```

(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)      ▷ instantiate an executable graph
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).

## 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 [22] (CPU-based) and GKSG [9] (GPU-based). Table 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. 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 (CPU-based) | $O(l_{mt} \times \frac{\|V\|}{T_{mt}})$ | $O(l_{mt} \times np_{mt} \times k \times \frac{\|V\|}{T_{mt}})$ |
| GKSG (GPU-based) | $O(l_{GKSG} \times \frac{\|V\|}{T_{GKSG}})$ | $O(l_{GKSG} \times it_{GKSG} \times (k \times \frac{\|V\|}{T_{GKSG}} + \frac{B \times log(B)}{T_{GKSG}}))$ |
| G-kway (GPU-based) | $O(l_{gk} \times \frac{\|V\|}{T_{gk}})$ | $O(l_{gk} \times it_{gk} \times (k \times \frac{\|V\|}{T_{gk}} + \frac{B \times log(B)}{T_{gk}}))$ |

### 4.1 Coarsening Stage Comparison

During the coarsening stage, the coarsening algorithm reduces the graph size level by level until it falls below the coarsening threshold, $\gamma$. 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 $\gamma$.

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.

## 4.2 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 $it_{GKSG}$ is the number of refinement iterations. Since $T_{GKSG}$ is much larger than $T_{mt}$, 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., $it_{gk} \ll it_{GKSG}$), leading to a substantial speedup in the uncoarsening stage.

## 5 Experimental Evaluation

We evaluate the performance of G-kway on six industrial circuit graphs (pci_bridge, vga_lcd, wb_dma, usb, tv80, and mem_ctrl) generated by [13–15], where regular graphs are used to represent timing graphs. Additionally, we test G-kway's performance on four large non-circuit graphs (ldoor, NLR, delaunay, and asia.osm) from DIMACS Graph Partitioning Challenge [33] to demonstrate our applicability beyond CAD algorithms. Table 2 lists the number of vertices and edges of each graph. We implement G-kway using C++17 and CUDA 12.4 and compile it with nvcc on a host compiler of GCC-8 with -O3 enabled. We run 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.

Table 2. Benchmark Sizes

| Name | # Vertices | # Edges |
|---|---|---|
| pci_bridge | 12,394,539 | 15,809,551 |
| vga_lcd | 13,923,210 | 24,904,499 |
| wb_dma | 19,686,000 | 20,236,297 |
| usb | 25,215,939 | 31,630,268 |
| tv80 | 13,102,222 | 17,759,671 |
| mem_ctrl | 6,422,461 | 8,455,835 |
| ldoor | 952,203 | 22,785,136 |
| NLR | 4,163,763 | 12,487,976 |
| delaunay | 16,777,216 | 50,331,601 |
| asia.osm | 11,950,757 | 12,711,603 |

Table 3. Overall comparison of runtime (second) 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. The cut size improvement over one shows that G-kway finds a better cut size than the competitor.

| Benchmark | GKSG | | mt-metis | | G-kway | | Speedup vs | | Cut Size Improvement vs | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Time (s) | Cut Size | Time (s) | Cut Size | Time (s) | Cut Size | GKSG | mt-metis | GKSG | mt-metis |
| pci_bridge | 0.89 | 5,114 | 3.21 | 4,773 | **0.27** | **4,293** | 3.5× | 12.5× | 1.2 | 1.1 |
| vga_lcd | 1.33 | 5,661 | 3.80 | 17,237 | **0.46** | **4,737** | 2.9× | 8.4× | 1.2 | 3.6 |
| wb_dma | 1.21 | 7,131 | 3.81 | 7,844 | **0.32** | **6,915** | 4.6× | 14.3× | 1.0 | 1.1 |
| usb | 2.03 | 7,933 | 6.97 | 10,283 | **0.58** | **6,709** | 3.5× | 12.0× | 1.2 | 1.5 |
| tv80 | 0.70 | 2,575 | 2.46 | 3,094 | **0.26** | **2,457** | 2.6× | 9.3× | 1.0 | 1.3 |
| mem_ctrl | 0.62 | 7,368 | 1.35 | 7,126 | **0.26** | **6,976** | 2.4× | 5.3× | 1.1 | 1.0 |
| ldoor | 0.84 | 25,676 | 0.21 | **25,088** | **0.13** | 25,578 | 6.5× | 1.6× | 1.0 | 1.0 |
| NLR | 0.16 | 4,432 | 0.34 | **4,262** | **0.15** | 4,705 | 1.1× | 2.3× | 0.9 | 0.9 |
| delaunay | 0.48 | 12,650 | 2.23 | 8,614 | **0.27** | **8,463** | 1.8× | 8.3× | 1.5 | 1.0 |
| asia.osm | 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 |

### 5.1 Baselines

We consider mt-metis v0.7.2 [22] and GKSG [9] as baseline partitioners. Mt-metis is a state-of-the-art CPU-parallel graph partitioner that renovates the sequential Metis algorithm [20] 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. We set the imbalance ratio ($\epsilon$) to 0.03 and the coarsening threshold ($\gamma$) to $\frac{|v|}{20 \times log_2(k)}$. All data is an average of ten runs.

## 5.2 Overall Performance Comparison

Table 3 compares the overall runtime and cut size results among G-kway, GKSG, and mt-metis at $k = 2$. We run 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 coarsen 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.

## 5.3 Runtime Analysis

Figure 9 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 $k = 32$.

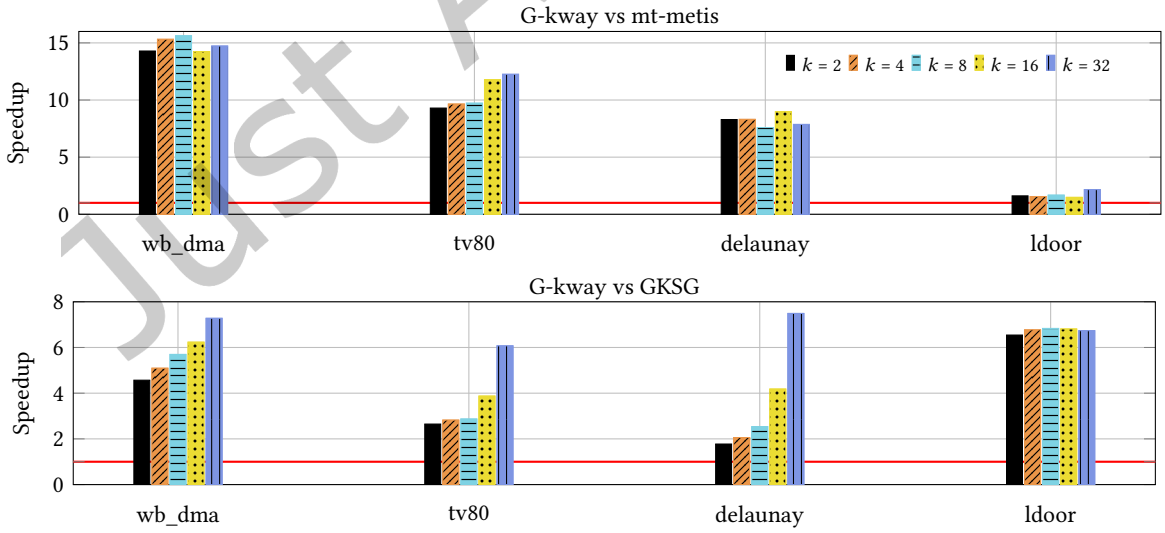


Fig. 9. The speedup of G-kway over mt-metis (top) and GKSG (bottom) at different $k$.

## 5.4 Cut Size Analysis

Figure 10 shows the cut size improvement ratio of G-kway over mt-metis (32 threads) and GKSG at $k = \{2, 4, 8, 16, 32\}$. In general, G-kway can produce partitions 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.

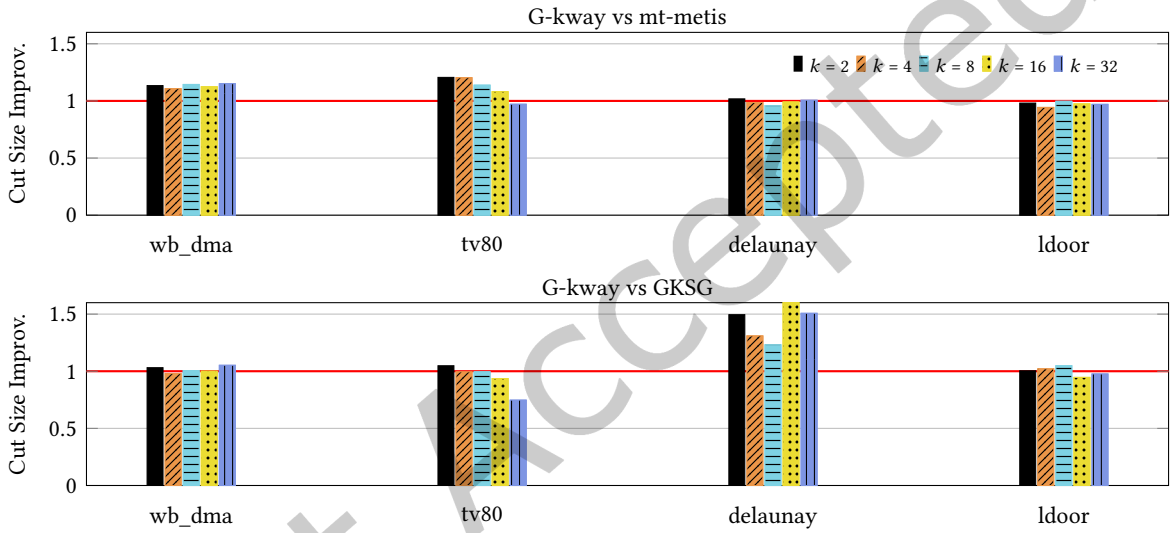


Fig. 10. The cut size improvement ratio of G-kway over mt-metis (top) and GKSG (bottom) at different $k$.

## 5.5 Absolute Efficiency over mt-metis

Figure 11 shows the speedup of G-kway over mt-metis using 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 12 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 add 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.

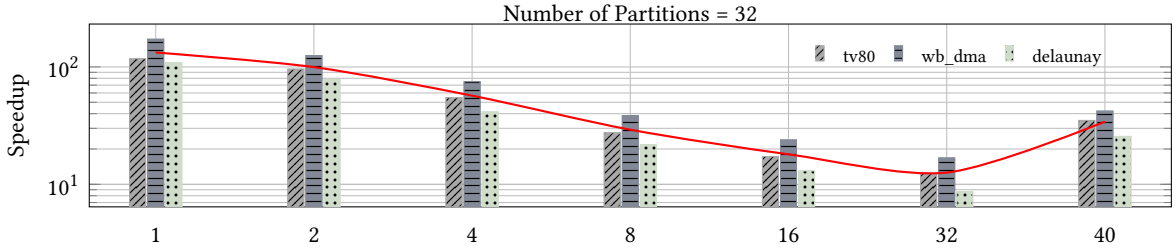


Fig. 11. The speedup of G-kway over mt-metis at various numbers of threads for tv80, wb_dma, and delaunay at $k = 32$. The red line indicates the average speedup trend of the three graphs.

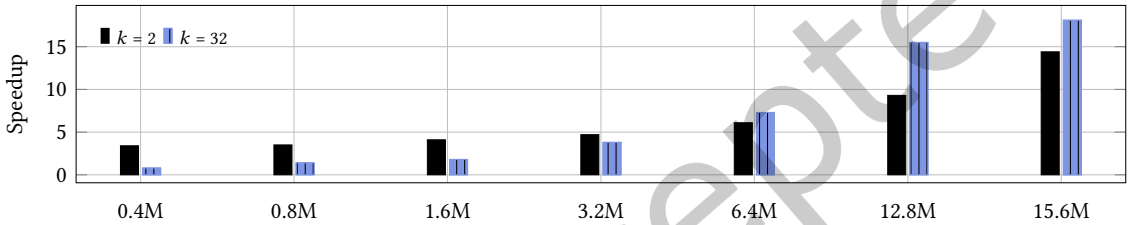


Fig. 12. The speedup of G-kway over mt-metis at varying graph sizes modified from usb at $k = 2$ and $k = 32$.



## 5.6 Analysis of Coarsening with and without Scoring

Table 4 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$^{-s}$ group many vertices into the same subset, resulting in a highly imbalanced coarsened graph. Such imbalance greatly impacts the partition results at later initial partition and refinement stages.

Table 4. 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%**) |

## 5.7 Analysis of Coarsening Threshold on Partitioning Performance

Table 5. Cut size and runtime (in seconds) comparisons with three coarsening threshold ( $\gamma$): $k$, $160 \times k$, 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 | $k = 2$ | | | | | | $k = 32$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\gamma = k$ | | $\gamma = 160 \times k$ | | $\gamma = \frac{\lvert V \rvert}{20 \times log_2(k)}$ | | $\gamma = k$ | | $\gamma = 160 \times k$ | | $\gamma = \frac{\lvert V \rvert}{20 \times log_2(k)}$ | |
| | 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 5 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 ($\gamma$): $k$, $160 \times k$, and $\frac{|V|}{20 \times log_2(k)}$. All data is an average of ten runs.

When $\gamma$ 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 $\gamma$ 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 $\gamma$ 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 $\gamma$ depends on the input graph size. For large graphs, $\gamma$ can become significantly large, making the initial partitioning computationally expensive. To achieve the best performance, we set $\gamma$ to $160 \times k$ for the rest of our experiments.

## 5.8 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 6 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 13 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 14 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 6. 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 |

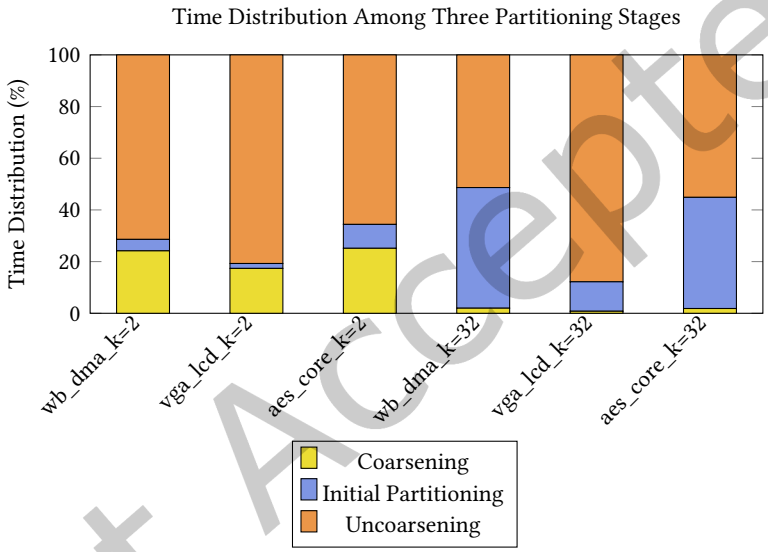


Fig. 13. Time distribution among the coarsening, initial partitioning, and uncoarsening stages for the three largest circuit graphs—wb_dma, vga_lcd, and aes_core—at $k$ = 2 and $k$ = 32.

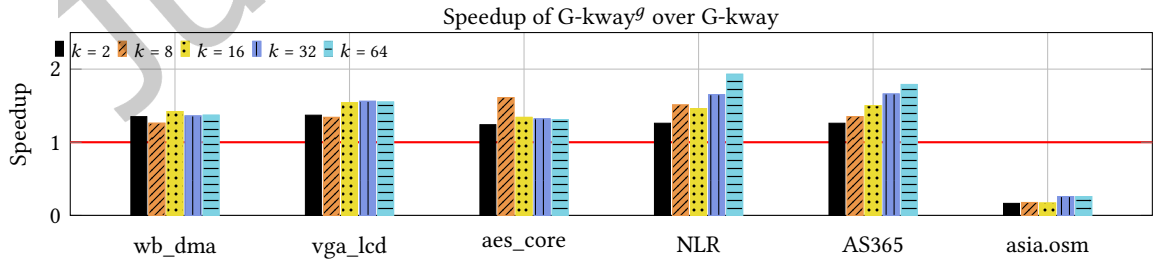


Fig. 14. The speedup of G-kway with CUDA Graph-based uncoarsening, G-kway$^g$, over the default G-kway during the uncoarsening stage at $k$ = {2, 8, 16, 32, 64}.

## 5.9 CUDA Graph Time Breakdown

Figure 15 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.

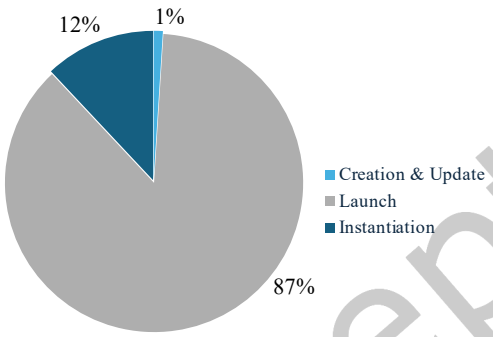


Fig. 15. The breakdown of total graph execution time spent on each operation, including creation & update, launch, and instantiation, for vga_lcd in G-kway$^{g}$.

Table 7. The maximum number of host calls for graph creation, instantiation, and launch for G-kway$^{g}$ and G-kway$^{g,-cond}$ was observed across ten runs for each value of k = {2, 8, 16, 32, 64}. Both G-kway$^{g}$ and G-kway$^{g,-cond}$ 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$^{g,-cond}$ omits the conditional node, relying on the host for control flow management.

| Benchmark | G-kway$^{g}$ | | | G-kway$^{g,-cond}$ | | |
|---|---|---|---|---|---|---|
| | 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 |



## 5.10 Analysis of CUDA Graph with and without CUDA Graph Conditional Nodes

Table 5 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$^{g,-cond}$ 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 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 8. Cut size and runtime (in seconds) comparisons with varying imbalance ratio ($\epsilon$) 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 |

## 5.11 Impact of Imbalance Ratio on Partitioning Performance

Table 8 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 $\epsilon$: 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.

# 6 Conclusion

In this paper, 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. As part of our on-going work [23, 24], we plan to accelerate hypergraph partitioning using CUDA Graph.

## Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. This research is also conducted by ACCESS – AI Chip Center for Emerging Smart Systems (supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government) and the JC STEM Lab of Intelligent Design Automation (funded by the Hong Kong Jockey Club Charities Trust).

## References


[1] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2020. High-quality shared-memory graph partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2710–2722.

[2] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent advances in graph partitioning*. Springer.

[3] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. 2024. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[4] Jingsheng Jason Cong and Joseph R Shinnerl. 2013. *Multilevel optimization in VLSICAD*. Vol. 14. Springer Science & Business Media.

[5] Charles M. Fiduccia and Robert M. Mattheyses. 1982. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*. IEEE, IEEE, 175–181.

[6] Miroslav Fiedler. 1975. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak mathematical journal* 25, 4 (1975), 619–633.

[7] Michael R Garey, David S Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*. 47–63.

[8] Bahareh Goodarzi, Martin Burtscher, and Dhrubajyoti Goswami. 2016. Parallel graph partitioning on a CPU-GPU architecture. In *IPDPSW*. IEEE.

[9] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High performance multilevel graph partitioning on GPU. In *HPCS*. IEEE.

[10] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep Multilevel Graph Partitioning. In *29th Annual European Symposium on Algorithms, ESA 2021 (LIPIcs, Vol. 204)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 48:1–48:17.

[11] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[12] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[13] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[14] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TPDS* 33, 6 (2022), 1303–1320.

[15] Tsung-Wei Huang and Martin DF Wong. 2015. OpenTimer: A high-performance timing analysis tool. In *ICCAD*. IEEE.

[16] Jan Hungershöfer and Jens-Michael Wierum. 2002. On the quality of partitions based on space-filling curves. In *International Conference on Computational Science*. Springer, 36–45.

[17] Laurent Hyafil and Ronald L Rivest. 1973. Graph partitioning and constructing optimal decision trees are polynomial complete problems. *(No Title)* (1973).

[18] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[19] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. 2011. *VLSI physical design: from graph partitioning to timing closure*. Vol. 312. Springer.

[20] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC* 20, 1.

[21] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *PACT*. IEEE.

[22] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *IPDPS*. IEEE.

[23] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, , and Tsung-Wei Huang. 2025. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

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

[25] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE DAC*.

[26] Dian-Lun Lin. 2024. *Task-parallel Heterogeneous Programming System for Logic Simulation*. Ph. D. Dissertation. The University of Wisconsin-Madison.

[27] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[28] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU computation using task graph parallelism. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer, 435–450.

[29] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 3041–3052.
[30] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[31] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ICPP*.
[32] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Brucek Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[33] Henning Meyerhenke and David A.. Bader. 2013. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop*. AMS.
[34] Burkhard Monien, Robert Preis, Stefan Schamberger, and T Gonzalez. 2007. Approximation algorithms for multilevel graph partitioning. *Handbook of approximation algorithms and Metaheuristics* 10 (2007), 1–60.
[35] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[36] NVIDIA. 2019. CUDA Graphs: Accelerating Your GPU Workloads. https://developer.nvidia.com/blog/cuda-graphs/. Accessed: 2024-08-23.
[37] Alex Pothen, Horst D Simon, Lie Wang, and Stephen T Barnard. 1992. Towards a fast implementation of spectral nested dissection. In *SC*. IEEE.
[38] Horst D Simon. 1991. Partitioning of unstructured problems for parallel processing. *Computing systems in engineering* 2, 2-3 (1991), 135–148.
[39] Vinod Grover Yan LIN. 2018. *Using CUDA Warp-Level Primitives*.
[40] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[41] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

Received 30 August 2024; revised 6 February 2025; accepted 23 April 2025