G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner

Wan Luan Lee
University of Wisconsin-Madison
Madison, USA
wanluan.lee@wisc.edu

Dian-Lun Lin
University of Wisconsin-Madison
Madison, USA
dianlun.lin@wisc.edu

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, USA
tsung-wei.huang@wisc.edu

Shui Jiang
The Chinese University of Hong Kong
Hong Kong, China
sjiang22@cse.cuhk.edu.hk

Tsung-Yi Ho
The Chinese University of Hong Kong
Hong Kong, China
tyho@cse.cuhk.edu.hk

Yibo Lin
Peking University
Beijing, China
yibolin@pku.edu.cn

Bei Yu
The Chinese University of Hong Kong
Hong Kong, China
byu@cse.cuhk.edu.hk

Abstract

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. 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. 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.

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 [7] 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 [7]. 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, [1] 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 [2] 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., 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 [2].

To overcome these problems, we propose G-kway, a new GPU-accelerated k-way graph partitioner. G-kway introduces a union find-based coarsening algorithm that can merge many vertices simultaneously to substantially reduce the number of coarsening levels while keeping good partitioning quality. Additionally, G-kway introduces a new independent set-based refinement algorithm that can refine many vertices in parallel, largely reducing the number of refinement iterations. To further optimize the performance of our GPU kernels, G-kway applies various modern GPU optimization techniques, such as pinned memory access and warp-level primitives.

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 [7] and GPU-based GKSG [2]. 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 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 $W_{p_i} \leq (1+\epsilon)\frac{\sum_{v \in V} W_v}{k}$, where $0 < \epsilon \ll 1$ and ϵ is the imbalance ratio given by applications.

3 GPU Multilevel k-way Partitioner

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 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 [5] 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 [2] for efficient GPU computing.

3.1 Union Find-based Coarsening with Scoring

Most existing parallel multilevel graph partitioners such as GKSG [2] 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
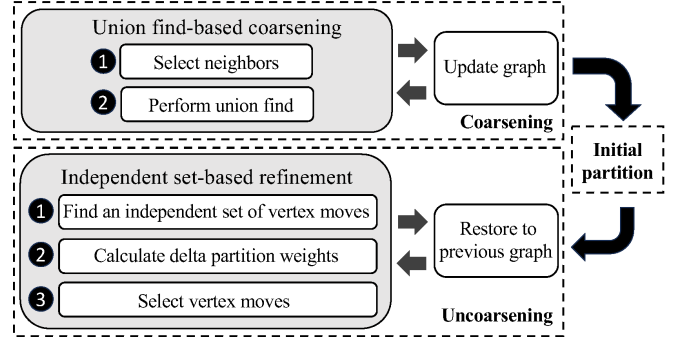

Figure 1: Overview of G-kway that consists of three main stages: coarsening, initial partition, and uncoarsening.

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 (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.
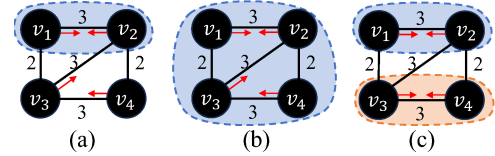

Figure 2: Examples of three coarsening methods for one iteration, including (a) Heavy Edge Matching (HEM) by GKSG, (b) Union find-based coarsening without scoring, and (c) Union find-based coarsening with scoring. Each vertex has a red arrow pointing to its selected neighbor. Vertices circled in the 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 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 2 (c) shows our union find-based coarsening with scoring. $v3$ selects $v4$ instead of $v2$ since $v4$ has lower degree than $v2$, resulting in two balanced subsets. Our coarsening algorithm consists of two steps: *select neighbors* and *perform union find*.

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 [6]. 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 [6]. 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 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.

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, 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).

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:         $s(u, v) \leftarrow$ c $\times W_{(u,v)}$ - 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:         $nbr \leftarrow selected\_nbr[v_i]$
16:         $v_i\_subset\_ID \leftarrow d\_subset\_ID[v_i]$
17:         $nbr\_subset\_ID \leftarrow d\_subset\_ID[nbr]$
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: **}**

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 [2]. 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.

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.

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**

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.

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 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 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 jth element in segment s stores the accumulated delta partition weight from the first to the jth vertex moves for partition s (i.e., a sub-sequence from the first to the jth 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.
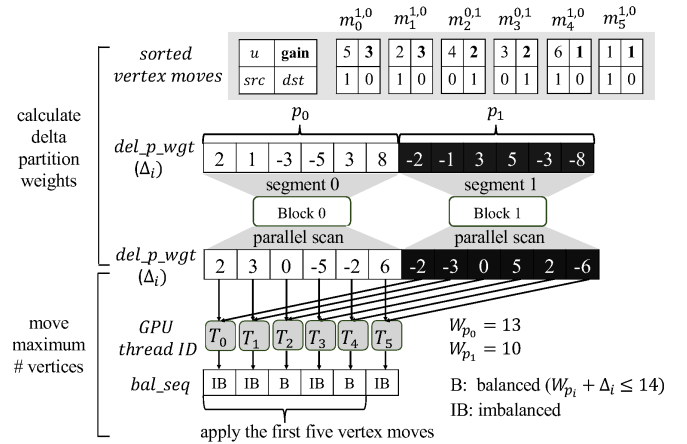

Figure 3: Illustration of the process to construct del_p_wgt and bal_seq with $k = 2$ under six vertex moves. Assuming current partition weights are 13 and 10 for p_0 and p_1, respectively, with a balance constraint of 14.

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 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: $seg\_size \leftarrow \min(\#vertex\_moves, 1024)$
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: }

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 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 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 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 $W_{p0} + \Delta_0 \leq 14$ 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 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.

4 Experimental Evaluation

We evaluate the performance of G-kway on six industrial circuit graphs generated by [3, 4], 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, asia.osm) from DIMACS Graph Partitioning Challenge to demonstrate our applicability beyond CAD algorithms. Table 1 lists the statistics of each graph. We implement G-kway using C++17 and CUDA 12.0 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.

4.1 Baselines

We consider mt-metis v0.7.2 [7] and GKSG [2] as baseline partitioners. Mt-metis is a state-of-the-art CPU-parallel graph partitioner that renovates the sequential Metis algorithm [5] 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 [7] and GKSG [2] that can produce the best results. All data is an average of ten runs.

4.2 Overall Performance Comparison

Table 1 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.

4.3 Runtime Analysis

Figure 4 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

Benchmark			GKSG		mt-metis		G-kway		Speedup vs		Cut size improvement 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: 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.
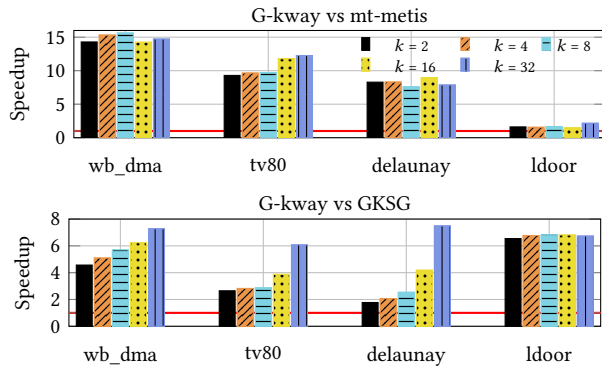

Figure 4: The speedup of G-kway over mt-metis (top) and GKSG (bottom) at different k.

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 $k = 32$.

4.4 Cut Size Analysis

Figure 5 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 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

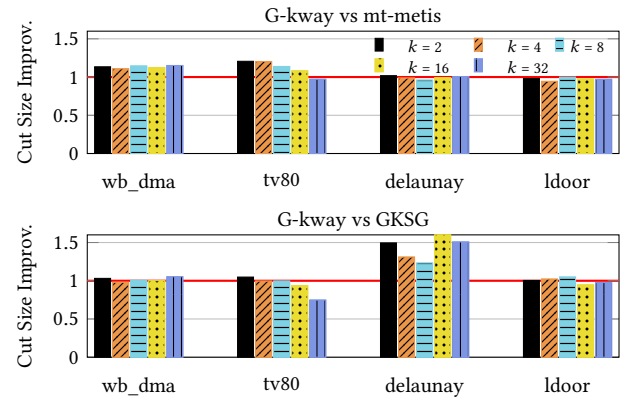vertices which can benefit from more fine-grained refinement as GKSG.

Figure 5: The cut size improvement ratio of G-kway over mt-metis (top) and GKSG (bottom) at different k.

5 Conclusion

In this paper, we have introduced G-kway, an efficient GPU-accelerated multilevel k-way graph partitioner. Experimental results have shown that G-kway outperforms the state-of-the-art CPU-based and GPU-based parallel partitioners.

References

[1] Bahareh Goodarzi, Martin Burtscher, and Dhrubajyoti Goswami. 2016. Parallel graph partitioning on a CPU-GPU architecture. In *IPDPSW*. IEEE.
[2] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High performance multilevel graph partitioning on GPU. In *HPCS*. IEEE.
[3] 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.
[4] Tsung-Wei Huang and Martin DF Wong. 2015. OpenTimer: A high-performance timing analysis tool. In *ICCAD*. IEEE.
[5] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC* 20, 1.
[6] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *PACT*. IEEE.
[7] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *IPDPS*. IEEE.