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

Wan Luan Lee[1], Shui Jiang[2], Dian-Lun Lin[1], Che Chang[1], Boyang Zhang[1],
Yi-Hua Chung[1], Ulf Schlichtmann[3], Tsung-Yi Ho[2], Tsung-Wei Huang[1]
[1]*University of Wisconsin–Madison, USA*
[2]*The Chinese University of Hong Kong, Hong Kong*
[3]*Technical University of Munich, Germany*
{wlee329, dianlun.lin, cchang289, bzhang523, yihua.chung, tsung-wei.huang}@wisc.edu,
{sjiang22, tyho}@cse.cuhk.edu.hk, ulf.schlichtmann@tum.de

***Abstract*—Recent advances in GPU-accelerated graph partitioning have achieved significant performance gains but remain limited to full graph partitioning, lacking support for incremental updates. This limitation is critical in CAD applications, where circuit graphs undergo iterative, incremental modifications during optimization. We present *iG-kway*, the first GPU-based incremental k-way graph partitioner. iG-kway features an incrementality-aware data structure and a refinement kernel that efficiently updates only affected vertices with minimal quality loss. Experiments show that iG-kway delivers up to 84× speedup over the state-of-the-art G-kway with comparable partitioning quality.**

I. Introduction

Graph partitioning is important for the design of efficient computer-aided design (CAD) algorithms because it allows an algorithm to break down a circuit graph into smaller and manageable pieces. However, as circuit graphs continue to grow in size, graph partitioning becomes increasingly time-consuming. For example, the sequential graph partitioner, metis [1], can take several minutes to partition just one million-node graph [2], and the runtime keeps increasing as the graph size becomes larger. To reduce the runtime, researchers have proposed various parallel graph partitioning algorithms [3]–[12]. For example, mt-metis [4] parallelized multi-level graph partitioning using multi-core CPUs, but the speedup is limited to only 8–16 CPU threads [13]. More recently, G-kway [13], Jet [2], and GKSG [14], [15] explored data parallelism from different stages of graph partitioning and offloaded time-consuming tasks (e.g., coarsening, refinement) to GPU, achieving significant speedup for large graphs.

In addition to partitioning an input graph once, which we refer to as *full graph partitioning* (FGP), *incremental graph partitioning* (IGP) is a critical enabler for many CAD applications that incorporate graph partitioning in a loop. For instance, the multi-input RTL simulator [16], [17] counts on iterative IGP to discover an optimal task graph for heterogeneous scheduling [18], [19]; similarly, a timing-driven optimizer also relies on iterative IGP to enhance the runtime performance of a timing analyzer [20]. In these applications, when the graph is incrementally modified, the partitioner must quickly refine the partitioning result to maintain a reasonable turnaround time across thousands or even millions of incremental iterations. As shown in Figure 1, IGP can save a significant amount of time compared to FGP. Without IGP, we cannot fully unlock the benefit of graph partitioning.

IGP has been studied in prior works [21]–[25], with a core focus on refining small subgraph regions affected by graph modifications, rather than performing a full re-partitioning from scratch. However, these works are largely limited to CPU architectures and can become inefficient when handling large graphs or when affected regions are large. Inspired by the recent success of GPU-accelerated FGP [13], we believe that GPU can also enhance the performance of IGP due to the large amount of data parallelism exhibited by graph partitioning. More importantly, as more CAD applications begin leveraging GPU acceleration [16], [17], [26]–[40], there is an increasing need to re-target time-consuming CPU tasks to GPU. For instance, a GPU-accelerated IGP will further speed up the incremental task graph optimization process of [16] (which takes several hours) while reducing the cost of moving and converting graph data between CPU and GPU during iterative IGP.

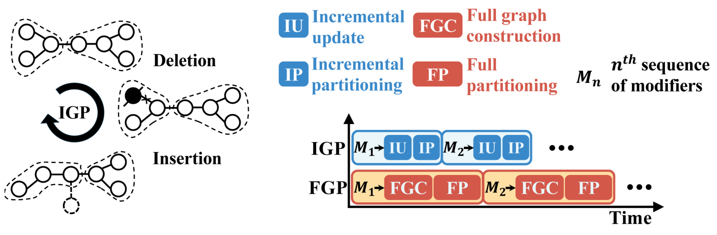
Deletion
IGP
Insertion
IU Incremental update
FGC Full graph construction
IP Incremental partitioning
FP Full partitioning
M_n n^{th} sequence of modifiers
IGP
M_1 IU IP M_2 IU IP •••
FGP
M_1 FGC FP M_2 FGC FP •••
Time

Fig. 1. Incremental graph partitioning (IGP) and its runtime advantage over full graph partitioning (FGP).

However, designing a GPU-parallel incremental graph partitioner is very challenging. First, existing GPU partitioners [2], [13], [14] count on *static 1D arrays*, such as the compressed sparse row (CSR), to store graphs on GPU. These static data structures make it very challenging to update graphs without rebuilding the entire arrays. Second, graph modifiers can affect vertices in different subgraph regions of different sizes. This varying number of affected vertices needs a clever strategy to balance workloads across GPU threads during incremental partitioning. Third, graph modifiers can potentially disrupt the balance of existing partitions. We need an effective GPU kernel algorithm to quickly restore partition balance while maintaining a satisfactory cut size.

While parallel IGP has been previously studied [24], [25], these efforts focused on CPU architectures. Due to the distinct performance models and memory hierarchies between CPU and GPU, we cannot directly apply these methods to GPU. For example, IOGP [24] introduced an online graph partitioning algorithm for distributed graph databases. However, IOGP focuses on optimizing data locality to minimize communication overhead in a distributed computing environment, which differs from our application focus. On the other hand, [25] formulated IGP into a two-layer linear programming (LP) proxy problem and solved it using multiple CPU threads. However, their methods require iteratively reformulating the proxy LP problem, which is inherently sequential and cannot scale to large IGP problems.

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

- We introduce a GPU-aware bucket-list graph representation that can efficiently handle graph modifiers without requiring any data structure rebuilds.
- We aggregate the affected vertices in a centralized buffer and dynamically assign GPU threads to process them, ensuring balanced workloads across the GPU threads.

- We design a GPU-parallel refinement kernel algorithm that efficiently re-balances partitions by moving affected vertices to a pseudo-partition and performing incremental refinement.

We have evaluated the performance of iG-kway on industrial circuit graphs and compared our results with the state-of-the-art GPU-accelerated graph partitioner, G-kway [13]. Experimental results show that iG-kway achieves an average speedup of 84× over G-kway, with comparable cut sizes.

II. 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 $v \in p_i$, and its external neighbors as $adj_{ext}(v) = \{u \in adj(v) \mid P(u) \neq P(v)\}$, and its internal neighbor as $adj_{int}(v) = \{u \in adj(v) \mid P(u) = P(v)\}$. We define the cut size as $\sum_{e=(u,v)\in E, P(u)\neq P(v)} W_e$. Cut size is widely used for evaluating the quality of a partition since it represents the interconnect complexity among partitions. The partition weight of p_i is defined as $W_{p_i} = \sum_{v \in p_i} W_v$.

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

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

where $W_{p_{max}}$ is the maximum allowable partition weight and ϵ is the imbalance ratio given by applications. Given a partitioned graph, the first goal of IGP is to apply a sequence of *graph modifiers* to the graph. Each modifier corresponds to a vertex insertion (M_u^+), a vertex deletion (M_u^-), an edge insertion ($M_{(u,v)}^+$), or an edge deletion ($M_{(u,v)}^-$). In most IGP applications [16], [20], [41], the number of modifiers is smaller than the graph size. The next goal of IGP is to efficiently refine the modified graph without starting from scratch while minimizing the cut size.

III. Overview of iG-kway

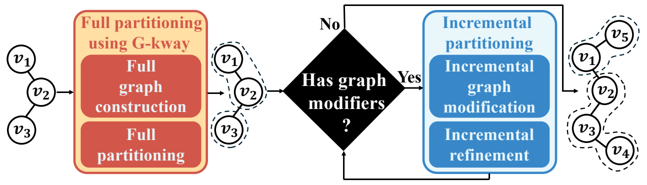

Fig. 2. Overview of the proposed incremental graph partitioner, iG-kway.

Figure 2 shows the overview of our GPU-parallel incremental k-way graph partitioner, iG-kway, which consists of two main stages: *full partitioning* (Section IV) and *incremental partitioning* (Section V). The goal of the full partitioning is to derive a high-quality partition from the original graph, which provides a foundation for the incremental partitioner to optimize subsequently modified graphs. We utilize G-kway [13] with a new constrained coarsening strategy to achieve a high-quality partitioning result. On the other hand, the goal of incremental partitioning is to update and refine the modified graph without starting from scratch. Our incremental graph partitioning has two main stages, *incremental graph modification* and *incremental refinement*, where (1) the former introduces a bucket-list data structure that stores the input graph and supports direct modification on GPU and (2) the latter introduces a GPU kernel algorithm that balances and refines the partition after the graph is incrementally modified.

IV. Full Partitioning with Constrained Coarsening

We use the state-of-the-art GPU-accelerated multilevel graph partitioner, G-kway [13] to perform full partitioning due to its high partitioning quality and fast runtime. G-kway employs a multilevel approach, iteratively coarsening the graph into a smaller representation until it reaches a manageable size, at which point the partitioning stage begins. To accelerate the coarsening process, G-kway employs a union-find-based coarsening that merges many vertices simultaneously to reduce the graph size per iteration. Here, G-kway introduces a parallel union-find algorithm to iteratively group vertices into subsets. Despite parallelism, this approach may result in imbalances in coarsened vertex weights, as each subset contains a varying number of vertices. This imbalance makes it challenging for the subsequent partitioning stage to achieve a balanced partition. Figure 3 (a) shows an example of G-kway's union-find-based coarsening, where G-kway groups vertices into two imbalanced subsets.

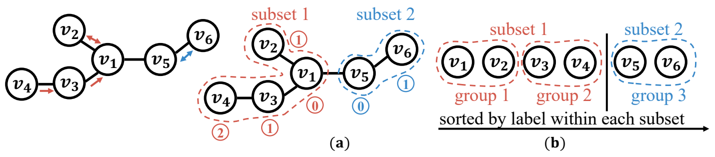

Fig. 3. Examples of two coarsening methods, including (a) G-kway's union-find based coarsening and (b) our constrained coarsening. Each vertex has an arrow pointing to its selected neighbor, and its label ⓝ or ⓝ indicates the iteration when it was grouped into the subset. Vertices circled in the red dashed line will be coarsened into a coarsen vertex.

To address this issue, we divide the vertices in each subset into small, fixed-size groups of size s and merge the vertices within each group into a single coarsened vertex in parallel. This strategy reduces the graph size in each iteration by merging multiple vertices simultaneously while maintaining balanced coarsened vertex weights. To this end, a straightforward solution is randomly selecting s vertices to form smaller groups. However, this approach may result in poor partitioning quality, as vertices that are far apart could end up in the same group and be merged into a single coarsened vertex, distorting the original graph structure. For instance, in Figure 3 (a), randomly dividing vertices in subset 1 can result in placing distant vertices, v_2 and v_4, in the same group.

To address this problem, we modify G-kway's union-find-based coarsening algorithm by labeling each vertex with the iteration in which it joins a subset. Since vertices that are farther apart are merged into the subset in later iterations, we can sort the vertices based on their labels and divide them into groups, ensuring vertices that are closer together are placed in the same group. Figure 3 (b) shows our *constrained coarsening* strategy, where vertices are first sorted by iteration number in ascending order within groups, and large subsets are divided into smaller groups of size two. In this approach, groups of similar size merge into a single coarsened vertex, producing more balanced coarsened vertex weights.

V. Incremental Partitioning

A. Bucket-list Graph Representation

Existing GPU graph partitioners [2], [13], [14] count on CSR to store graphs on GPU. In CSR, all vertices' neighbors are concatenated into an adjacency list of size $|E|$, with an adjacency pointer recording each vertex's neighbor position. However, this statically packed data structure makes modifying the graph very challenging without rebuilding the structure. For instance, inserting an edge in the CSR data structure requires shifting all elements in the adjacency list and updating their adjacency pointers. Furthermore, updating the CSR is typically done on the CPU, which introduces additional data movement and conversion overhead between the CPU and GPU.

To overcome this challenge, we propose a bucket-list data structure that supports efficient graph modifications directly on the GPU by storing each vertex's neighbors in pre-allocated buckets. We design the buckets with 32 slots to align with the GPU warp size (32 threads), reducing thread divergence and enabling efficient intra-warp communication using CUDA warp-level primitives [42]. To avoid rebuilding the bucket-list data structure, we allocate extra buckets for each vertex to accommodate future edge insertions. The number of buckets for vertex u is determined by the following formula:

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

where $D(u)$ is the degree of u (i.e., the number of neighbors of u) and γ is the number of the extra buckets per vertex (by default, iG-kway sets γ to one). To accommodate bigger graph modifications, applications can set a higher γ. Figure 4 (a) shows an example of our bucket-list data structure, where each bucket contains four slots. All buckets are concatenated into a bucket-list, with a bucket pointer recording each vertex's bucket position within the list. To avoid reallocating memory when more buckets are required (e.g., due to vertex insertion), we pre-allocate a large block of memory for the bucket-list and use a pointer to track the current number of buckets.

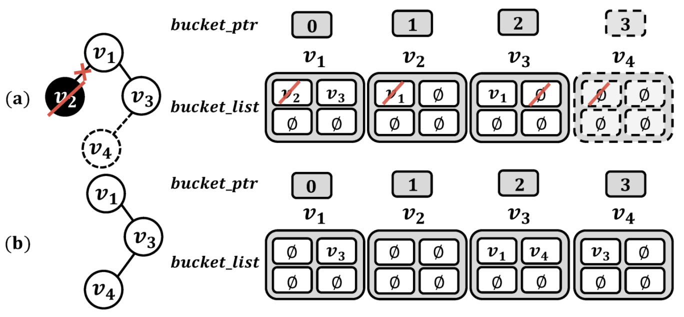

Fig. 4. An example of our bucket-list data structure, where each vertex has a bucket, and each bucket contains four slots. Empty slots are denoted by ∅. Figures (a) and (b) show the graph before and after applying the following graph modifiers: $M^-_{v_2}$, $M^+_{v_4}$, $M^-_{(v_2,v_1)}$, $M^-_{(v_1,v_2)}$, $M^+_{(v_4,v_3)}$, and $M^+_{(v_3,v_4)}$.

B. Incremental Graph Modification

1) Edge Modifiers: To efficiently handle $M^+_{(u,v)}$ (insert an edge (u,v)), we use a GPU warp to locate an empty slot in vertex u's buckets through fast intra-warp communication and insert vertex v to the first empty slot. Algorithm 1 presents our edge insertion algorithm. All threads in the warp first fetch the number of buckets allocated to u and the start position of its buckets from *bucket_ptr* (lines 4-5). Threads then process one bucket at a time until an empty slot is found or all buckets are checked (line 6). To efficiently locate an empty slot, each thread fetches a slot value in the bucket and uses the warp-level primitive [42], `__ballot_sync`, to simultaneously evaluate whether its slot is empty, storing the results in a bitmask (lines 7-8). If an empty slot is found, `__ffs` returns the first empty slot (line 9), and threads insert the edge into this slot and then terminate. (lines 10-12). If no slot is empty, each thread increments the bucket counter and continues looking for an empty slot in the next bucket (line 13).

We handle $M^-_{(u,v)}$ using the same strategy as Algorithm 1 except that instead of locating an empty slot, threads within the warp work simultaneously to find v in u's buckets and mark v as empty.

2) Vertex Modifiers: To delete a vertex u, a straightforward approach is to remove its buckets from the bucket-list. However, this approach can incur significant overhead, as it needs to recalculate the bucket pointer and rebuild the bucket-list. To address this problem, we use a vertex status array to track each vertex's current status (deleted or active) without removing its buckets from the bucket-list. Algorithm 2 presents our approach for handling the vertex modifier (M^+_u and M^-_u) using a GPU warp. To handle M^-_u, threads within the same warp first mark u as deleted in the vertex status array (line 5) and cooperatively remove all of u's neighbors by marking all slots in its buckets as empty (lines 11-13). Similarly, to handle M^+_u, threads within the same warp first mark u as active in the vertex status array (line 8). Then, they assign u a single bucket and add the bucket to the end of the bucket-list by updating the bucket pointers accordingly (lines 9-10). Finally, threads initialize all slots in u's buckets as empty (lines 11-13).

Algorithm 1: Edge Insertion

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

Algorithm 2: Vertex Insertion / Deletion

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

C. Incremental Refinement

Once the graph is incrementally modified, the existing partitioning result may become invalid due to the change in graph structure and balance condition. For example, adding too many vertices can cause the partition to violate the balance constraint, while inserting edges may require relocating vertices to reduce the cut size. To refine a modified graph without starting from scratch, one possible approach is to use the independent-set-based refinement algorithm in G-kway [13]. However, G-kway's refinement algorithm does not account for potential imbalances that may occur after applying graph modifiers. Additionally, because it lacks IGP support, G-kway refines all vertices on the partition boundary (i.e., vertices with $adj_{ext} \neq 0$), which is unnecessary when only local subgraph regions are affected. To address this problem, we propose an efficient incremental refinement algorithm that restores partition balance and refines only the vertices affected by graph modifiers. Our algorithm consists of two steps, *partition balancing* and *parallel refinement*, explained below:

Algorithm 3: Partition Balancing

```
   Input: bucket_ptr, bucket_list, partition, vertex_in_pseudo,
          vertex_in_pseudo_size
   Input: affected_vertex initialize to false
   // assign a graph modifier to a GPU warp
1  parallel for each thread in the GPU warp
2      if vertex_insertion $M_u^+$
3          partition[u] ← pseudo
4          pos ← atomicAdd(vertex_in_pseudo_size, 1)
5          vertex_in_pseudo[pos] = u
6      else if edge_insertion $M_{(u,v)}^+$ or edge_deletion $M_{(u,v)}^-$
7          affected_vertex[u] ← true ; affected_vertex[v] ← true
   // assign each u in affected_vertex to a GPU warp
8  parallel for each thread in the GPU warp
9      if partition[u] == pseudo
10         return
11     lane_id ← thread index in the GPU warp
12     bucket_cnt ← 0
13     bucket_start ← bucket_ptr[u]
14     num_bucket ← bucket_ptr[u + 1] - bucket_start
15     cur_par ← partition[u]
16     while bucket_cnt < num_bucket
17         nbr ← bucket_list[bucket_start + bucket_cnt × 32 + lane_id]
18         nbr_par ← nbr == ∅? ∅ : partition[nbr]
19         $adj_{ext}$ += __popc(__ballot_sync(FULL, nbr_part != cur_par
             && nbr != ∅))
20         $adj_{int}$ += __popc(__ballot_sync(FULL, nbr_part == cur_par))
21         bucket_cnt++
22     if $adj_{ext}$ > $adj_{int}$ && lane_id == 0
23         pos ← atomicAdd(vertex_in_pseudo_size, 1)
24         vertex_in_pseudo[pos] = u
   // assign u in vertex_in_pseudo to a GPU thread
25 parallel for each GPU thread
26     partition[u] ← pseudo
```

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

A vertex is considered affected by graph modifiers if: (1) it has been directly modified (e.g., its edges have been deleted or inserted), or (2) its neighbors have been modified, as changes in their surrounding structure may require refinement. However, not all affected vertices require refinement. If a vertex has as many or more adj_{int} than adj_{ext}, moving it to another partition will not reduce the cut size and may even increase it. To avoid this, we filter out such vertices and do not move them to the pseudo-partition. We use a centralized buffer, $vertex\_in\_pseudo$, to store vertices in the pseudo-partition, as they are often scattered across different parts of the graph, making it challenging to balance the workload across GPU threads. By aggregating these vertices to the buffer, we can dynamically assign GPU threads to handle them, ensuring balanced workloads and significantly increasing GPU performance.

Algorithm 3 presents our partition balancing algorithm. We use an array, $affected\_vertex$, of size $|V|$ to record whether a vertex is affected. We assign each graph modifier to a GPU warp to mark directly modified vertices as affected. Threads assigned to M_u^+ move the vertex to the pseudo-partition immediately by changing its partition in $partition$ array, which records the partition assignment of each vertex, and insert it to $vertex\_in\_pseudo$ (lines 2-5). On the other hand, threads handling $M_{(u,v)}^+$ or $M_{(u,v)}^-$ mark u and v as affected (lines 6-7). Then, we check if they can be filtered out. To do this, we assign each vertex in $affected\_vertex$ to a GPU warp. Threads assigned to vertices already in the pseudo-partition terminate early (lines 9-10), while the remaining threads process one bucket at a time, with each thread fetching a neighbor and its corresponding partition (lines 17-18). Threads efficiently calculate external and internal neighbors using the warp-level primitive `__ballot_sync` to evaluate each neighbor's partition, storing the results in a bitmask and counting them with the warp-level primitive `__popc` (lines 19-20). They then continue processing the next bucket until all are completed. Next, the first thread in the warp checks if the vertex has fewer adj_{int} than adj_{ext} (line 22). If so, it adds the vertex to $vertex\_in\_pseudo$, without immediately updating its partition in the $partition$ array (lines 23-24).

The proposed strategy in Algorithm 3 prevents data races by deferring partition updates until threads in other warps have completed their calculations. Once all affected vertices have been processed, we launch another GPU kernel to update the partitions of vertices in $vertex\_in\_pseudo$. We then move vertices with affected neighbors by assigning each vertex u in the pseudo-partition to a GPU warp. Threads in the same warp mark all neighbors of u as affected, using the same approach to filter out the neighbors whose adj_{ext} is less than adj_{int} and move the rest to the pseudo-partition.

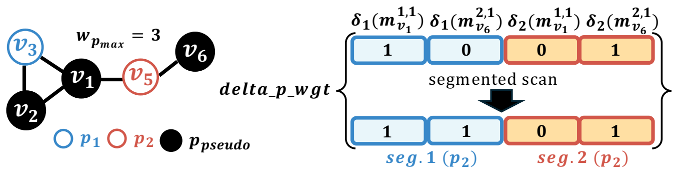

Fig. 5. Illustration of constructing $delta\_p\_wgt$ to calculate the accumulated delta partition weights for two partitions, p_1 and p_2, across two vertex moves using segmented scan. All vertices have a weight equal to one.

2) Parallel Refinement: Once the partition is balanced and affected vertices are in the pseudo-partition, we refine them by moving each vertex to its most suitable partition in parallel. We define the most suitable partition for a vertex u as the partition to which moving u from the pseudo-partition introduces the smallest cut size (i.e., the partition with most of u's neighbors), while maintaining the balanced partition. However, moving adjacent vertices in parallel requires costly synchronization to determine the most suitable partition accurately [13]. For example in Figure 5, if vertices v_1 and v_2 move concurrently from the pseudo-partition to other partitions, v_1 may initially select either p_1 or p_2 as its most suitable partition. However, if v_2 moves to p_1, v_1 should update its choice to p_1. Without synchronization between v_1 and v_2, v_1 can select the most suitable partition incorrectly. To overcome this synchronization challenge, we move non-adjacent vertices from $vertex\_in\_pseudo$ in parallel. For each non-adjacent vertex u, we create a vertex move $m_u^{par,\#nbr}$, where par is u's most suitable partition, and $\#nbr$ is the number of neighbors in par, and insert the vertex move to the $vertex\_moves$ buffer to form a sequence of vertex moves.

Algorithm 4 presents our parallel refinement algorithm. To find non-adjacent vertices from $vertex\_in\_pseudo$, we assign each vertex u in $vertex\_in\_pseudo$ to a GPU warp. Threads in the warp cooperatively process the bucket of u one at a time, with each thread fetching a neighbor and its partition (lines 6-7). Threads then efficiently check if any thread has a neighbor in the pseudo-partition with a vertex ID less than u's vertex ID using `__any_sync` (line 8). If such a neighbor exists, threads terminate early, as u is not selected to move because its neighbor is being moved in this iteration (lines 9-10). Threads that do not find such neighbors then continue to check neighbors stored in other buckets. Next, threads with a vertex selected to move cooperatively identify the most suitable partition for u by counting u's neighbors in each partition p with a weight that does not exceed $W_{p_{max}}$. To accomplish this, each thread fetches a neighbor stored in a bucket along with its partition and uses the same strategy (i.e., warp-level primitive) as in Algorithm 3 to efficiently count neighbors belonging to p (lines 15-19). After processing all buckets, the threads update max_nbr and max_nbr_p, stored in shared memory, if either 1) the number of neighbors in partition p exceeds the current maximum max_nbr, or 2) p has the same number of neighbors as max_nbr but a lower partition weight (line 20). Finally, the first thread in the warp creates a vertex move and inserts it to the $vertex\_moves$ (lines 21-23).

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

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

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

Figure 5 illustrates the calculation of accumulated delta partition weights for two partitions, performed using a parallel segmented scan with two vertex moves. Each element in $delta\_p\_wgt$ records the delta partition weight for a partition after applying a vertex move. The first two elements (i.e., segment 1) correspond to partition p_1, while the last two elements (i.e., segment 2) correspond to partition p_2. For example, $delta\_p\_wgt[0]$ records the partition weight change for the first vertex move in partition p_1. We then perform a parallel segmented scan on $delta\_p\_wgt$ to accumulate these changes for each partition. After the scan, the s^{th} element in each segment holds the accumulated change in partition weight from the first to the s^{th} vertex move in each partition, representing the accumulated weight change over this subsequence. We then identify the longest subsequence of vertex moves that satisfies the balance constraint to apply as many vertex moves as possible in parallel. In this example, both vertex moves can be applied, as neither p_1's partition weight plus $delta\_p\_wgt[1]$ nor p_2's plus $delta\_p\_wgt[3]$ exceeds W_{pmax}. We then repeat the same process until all vertices are moved from the pseudo-partition.

Algorithm 4: Parallel Refinement

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

VI. Experimental Evaluation

We evaluated the performance of iG-kway using seven industrial circuit graphs generated by [13], [43]. Additionally, we tested iG-kway on three large non-circuit graphs (coAuthorsCiteseer, adaptive, and NLR) from the DIMACS Graph Partitioning Challenge [44] to demonstrate its applicability beyond CAD algorithms. Table I lists the statistics of each graph. In our experiment, we applied 100 incremental iterations based on the setting of TAU 2015 Incremental Timing Contest [41], where each iteration involves tens to hundreds of design modifiers that randomly remove/insert vertices and edges from/into the graph.

We consider G-kway [13], a state-of-the-art GPU-accelerated k-way graph partitioner, as our baseline. To have a fair comparison with G-kway and focus on incrementality, we replace its coarsening algorithm with our constrained coarsening to achieve better performance. Furthermore, since G-kway counts on the CPU to generate a graph CSR on the GPU, we modify the graph and regenerate its CSR for each incremental iteration, then apply G-kway to partition the modified graph. Hereafter, we refer to this baseline as *G-kway*†.

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

A. Overall Performance Comparison

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

In terms of partitioning time, iG-kway outperforms G-kway† across all graphs with an average speedup of 84×. This speedup is due to iG-kway's incremental refinement algorithm, which identifies and refines only the vertices affected by graph modifiers, eliminating the need for re-partitioning required by G-kway†. Regarding the cut size, iG-kway finds a cut size comparable to G-kway† for all graphs. We attribute this to the effectiveness of our incremental refinement algorithm, which first moves vertices to a pseudo-partition to restore balance, and then moves them again to reduce the cut size.

Figure 6 details the comparison of partitioning time and cut size over 100 incremental iterations for the circuit usb under two different values of k. At the first iteration, we do not observe a significant runtime difference between iG-kway and G-kway† since both are FGP. However,

TABLE I
OVERALL COMPARISON OF RUNTIME (MODIFICATION AND PARTITIONING) AND CUT SIZE BETWEEN IG-KWAY AND G-KWAY[†] AT $k = 2$. ALL TIMES ARE MEASURED IN SECONDS. A CUT SIZE IMPROVEMENT GREATER THAN ONE INDICATES IG-KWAY CAN FIND A BETTER CUT SIZE.

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

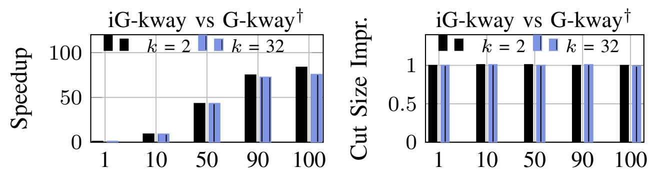

Fig. 6. Speedup (left) and cut size improvement (right) of iG-kway over G-kway[†] for the usb circuit over 100 incremental iterations.

as the number of incremental iterations increases, iG-kway's runtime advantage over G-kway[†] becomes more pronounced. The speedup of iG-kway grows proportionally with the number of incremental iterations for both k values. In terms of cut size, iG-kway achieves a comparable value to G-kway[†] across all iterations (within $\pm 3\%$).

B. Runtime and Cut Size Analysis under Varying k

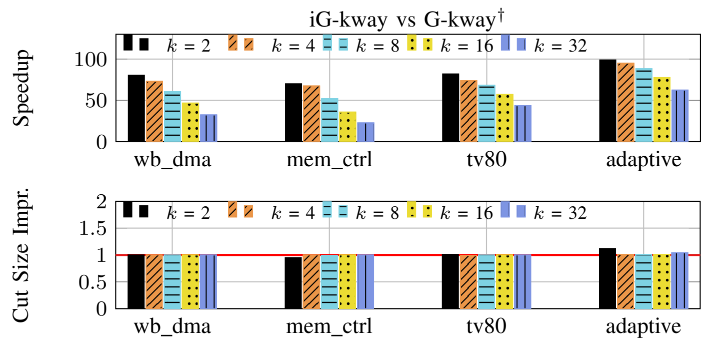

Fig. 7. The speedup (top) and cut size improvement (bottom) of iG-kway over G-kway[†] at different k values. A cut size improvement above one indicates that iG-kway can find a better cut size.

Figure 7 shows the speedup and cut size improvement of iG-kway over G-kway[†] at $k = \{2, 4, 8, 16, 32\}$ on three circuit graphs (wb_dma, mem_ctrl, and tv80) and a large non-circuit graph (adaptive). Regardless of k, iG-kway is consistently faster than G-kway[†]. iG-kway achieves up to a 98× speedup over G-kway[†] at $k = 2$. However, as k increases, the speedup decreases because iG-kway needs to examine more partitions to determine a suitable partition for each affected vertex. Despite this, iG-kway still achieves up to a 62× speedup at $k = 32$. Regarding the cut size, iG-kway achieves a comparable result with G-kway[†] on different k. These results highlight the effectiveness and efficiency of iG-kway over G-kway[†] in incremental graph partitioning.

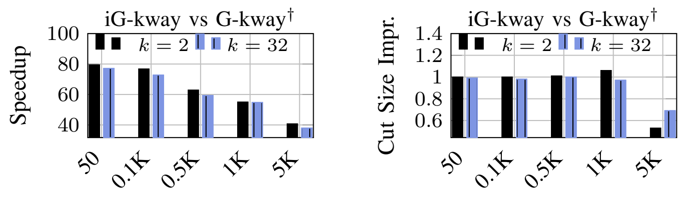

Fig. 8. The speedup (left) and cut size improvement (right) of iG-kway over G-kway[†] for the usb circuit across 100 incremental iterations each with varying numbers of graph modifiers (50–5K).

C. Incrementality Analysis

Figure 8 shows the speedup (left) and cut size improvement (right) achieved by iG-kway over G-kway[†] for the circuit usb across 100 incremental iterations each with varying numbers of graph modifiers (ranging from 50 to 5K) per iteration. When the number of graph modifiers per iteration is small, the advantage of iG-kway is more remarkable. For instance, with 50 graph modifiers, iG-kway is up to 80× faster than G-kway[†] while obtaining a comparable cut size. However, as the number of graph modifiers per iteration increases, the speedup decreases due to the growing number of affected vertices. More affected vertices require iG-kway to spend more time refining the partition. When the number of graph modifiers exceeds 5K per iteration, iG-kway struggles to find a partition with a decent cut size. This happens because after applying many graph modifiers (e.g., 5K×100 in this case), the graph becomes very different from its original form. This large difference makes it difficult for iG-kway to effectively optimize the graph, as its incremental refinement strategy relies on local graph structure. In such cases, applications can resort to FGP using G-kway[†], especially when the number of graph modifiers reaches 50% of the graph's size.

VII. CONCLUSION

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

ACKNOWLEDGMENT

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141, as well as the Research Grants Council of Hong Kong SAR (No. CUHK14207523).

References

[1] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

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

[3] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, "Parallel graph partitioning on multicore architectures," in *Languages and Compilers for Parallel Computing: 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers 23*. Springer, 2011, pp. 246–260.

[4] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 225–236.

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

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

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

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

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

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

[11] P. Sanders and D. Seemaier, "Brief announcement: Distributed unconstrained local search for multilevel graph partitioning," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 443–445. [Online]. Available: https://doi.org/10.1145/3626183.3660257

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

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

[14] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami, "High performance multilevel graph partitioning on gpu," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 769–778.

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

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

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

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

[19] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "Gcs-timer: Gpu-accelerated current source model based static timing analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3655983

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

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

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

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

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

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

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

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

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

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

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