iHyperG: Incremental Hypergraph Partitioning on GPU

Wan Luan Lee
wlee320@wisc.edu
University of Wisconsin-Madison
USA

Aditya Das Sarma
adassarma@cs.wisc.edu
University of Wisconsin-Madison
USA

Che Chang
cchang289@wisc.edu
University of Wisconsin-Madison
USA

Chih-Chun Chang
chih-chun.chang@wisc.edu
University of Wisconsin-Madison
USA

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

Abstract

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

ACM Reference Format:
Wan Luan Lee, Aditya Das Sarma, Che Chang, Chih-Chun Chang, and Tsung-Wei Huang. 2026. iHyperG: Incremental Hypergraph Partitioning on GPU. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803930

1 Introduction

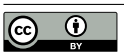
Hypergraph partitioning plays a key role in various stages of computer-aided design (CAD), including placement, routing, timing analysis, and logic simulation. For example, it helps optimize placement by dividing the circuit into smaller, more manageable blocks while minimizing interconnections among them [30]. However, as modern circuits continue to grow in size and complexity, hypergraph partitioning has become increasingly time-consuming. For instance, the widely used sequential partitioner hMetis can take several minutes to partition a circuit with only five million gates [19–21, 25].

To mitigate this runtime bottleneck, several parallel partitioning strategies have been developed. Among them, Mt-KaHyPar [7] is a CPU-based hypergraph partitioner that exploits multithreading to parallelize the coarsening and refinement stages, achieving significant speedups over the sequential hMetis [25]. More recently, HyperG [27] leverages the massive parallelism of GPU to accelerate the partitioning process even further, reporting up to 4× speedup over Mt-KaHyPar at 16 threads. Despite these runtime improvements, existing works focus only on *full hypergraph partitioning* (FHP), where the entire hypergraph is partitioned from scratch.

While FHP remains the dominant approach, many CAD applications benefit more from *incremental hypergraph partitioning* (IHP) where partitioning is integrated into iterative optimization loops. For example, a timing optimizer may repeatedly adjust cell placements to meet timing goals [18, 22], while logic synthesis tools incrementally restructure logic cones to improve design quality [38]. In these iterative optimization workloads, each time the circuit is incrementally modified, the partitioner must rapidly refine the partitioning result to maintain a reasonable turnaround time across thousands or even millions of incremental iterations. Without IHP, the overhead of repetitive FHP can accumulate significantly, and the benefits of hypergraph partitioning cannot be fully exploited.

While incremental graph partitioning has been studied on both CPU and GPU architectures [4–6, 24, 26, 40], IHP remains largely unexplored. However, inspired by the success of the GPU-accelerated incremental graph partitioner iG-kway [26], which refines affected vertices in parallel to achieve significant runtime improvements, we believe that IHP can similarly benefit from GPU due to the substantial data parallelism inherent in hypergraph partitioning. Moreover, as CAD applications increasingly adopt GPU acceleration [3, 9–16, 23, 28, 29, 31–37, 39, 41–43, 45–47], there is a growing need to retarget compute-intensive, CPU-bound tasks to GPU. For example, in a timing optimization flow that performs incremental placement across iterations to meet timing goals [18], a GPU-based IHP framework could significantly reduce runtime while minimizing the overhead of frequent CPU–GPU data transfers during iterative optimization.

Although iG-kway [26] has demonstrated effectiveness in supporting incremental graph partitioning on GPU, its strategy cannot be directly applied to hypergraphs due to fundamental structural differences between graphs and hypergraphs. For instance, to efficiently modify graphs on GPU, iG-kway employs a bucket-list data structure that trades higher memory usage for efficient handling of dynamic updates. However, adopting this data structure to hypergraphs results in excessive memory overhead and poor GPU scalability, as hyperedges require substantially more memory to represent complex multi-pin relationships. In addition to data structure limitations, hypergraphs also require a fundamentally different refinement strategy than graphs. For example, iG-kway refines all vertices whose incident edges have



DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803930

been modified, which is effective for graphs where each edge connects only two vertices. However, in hypergraphs, each hyperedge can connect many vertices across multiple partitions. Refining all vertices that are incident to modified hyperedges can be inefficient. Therefore, we need a hypergraph refinement strategy that identifies vertices requiring updates, reducing redundant computation while preserving partition quality.

To overcome these challenges, we introduce iHyperG, a GPU-parallel k-way hypergraph partitioner with efficient support for incremental updates. To the best of our knowledge, iHyperG is the first GPU solution that fully exploits parallelism for partitioning large-scale hypergraphs under frequent modifications. It features a delta-based hypergraph data structure that efficiently supports scalable incremental updates, an incremental refinement strategy that focuses on cut-critical vertices within modified hyperedges to reduce redundant computation while preserving partition quality, and a single-pass rebalancing algorithm that restores partition balance with minimal cut-size increase. Additionally, iHyperG leverages CUDA warp-level primitives for efficient intra-warp communication, enabling high-performance incremental hypergraph modification and partitioning.

We have evaluated the performance of iHyperG on industrial circuit designs and compared it against the state-of-the-art GPU-accelerated full hypergraph partitioner, HyperG [27]. Experimental results show that iHyperG achieves 190× and 83× speedups in modification and partitioning, while maintaining comparable cut sizes.

2 Problem Definition and Notation

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

The goal of FHP is to find a k-way partition from scratch that satisfies the balance constraint while minimizing $\text{cut}(P)$. The balance constraint limits the maximum weight of p_i as $W_{p_i} \leq W_{p_{\max}} = (1 + \epsilon)\frac{\sum_{v \in V} W_v}{k}$, $0 < \epsilon \ll 1$, where $W_{p_{\max}}$ is the maximum allowable partition weight and ϵ is the imbalance ratio. Given a partitioned hypergraph H, the first goal of IHP is to apply a sequence of *modifiers* to H. Each modifier is an operation on the vertex's and hyperedge's incidence. Specifically, an insertion modifier $M^{+}_{(v,e)}$ inserts vertex v into hyperedge e's incidence and e into v's incidence, while a deletion modifier $M^{-}_{(v,e)}$ removes v from e's incidence and e from v's incidence. In practice, the number of modifiers is small. IHP then refines the partition on the modified hypergraph without restarting from scratch, while maintaining balance and minimizing $\text{cut}(P)$.

3 Overview of iHyperG

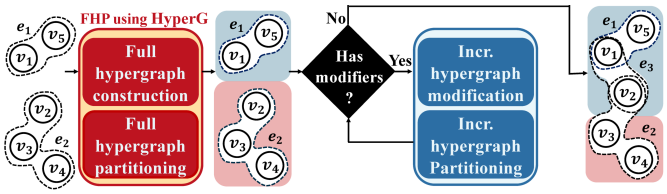


Figure 1: Overview of our incremental hypergraph partitioner.

Figure 1 shows the overview of our GPU-parallel incremental k-way hypergraph partitioner, iHyperG, which consists of two main stages: *full hypergraph partitioning* (FHP) and *incremental hypergraph partitioning* (IHP). The goal of the FHP is to derive a high-quality partition from the input hypergraph, providing a foundation for the incremental partitioner to optimize subsequently modified hypergraphs. We use HyperG [27], a state-of-the-art GPU-accelerated hypergraph partitioner, to achieve high-quality partitions.

On the other hand, the goal of IHP is to efficiently update and refine the partition of a modified hypergraph without starting from scratch. Our IHP framework has two main stages, *incremental hypergraph modification* (Section 4) and *IHP* (Section 5). During the incremental hypergraph modification stage, iHyperG employs a scalable delta-based hypergraph data structure to record updated incidences of modified vertices and hyperedges, avoiding a full rebuild of the hypergraph Compressed Sparse Row (CSR) data structure. During the IHP stage, iHyperG efficiently restores balance using a single-pass rebalancing algorithm and refines only cut-critical vertices, avoiding the need to repartition the hypergraph.

4 Incremental Hypergraph Modification

4.1 Delta-based Hypergraph Data Structure

Existing GPU hypergraph partitioners [27, 44] store hypergraphs on the GPU using a bidirectional CSR data structure with (i) a vertex to hyperedge incidence array `V2E` and (ii) a hyperedge to vertex incidence array `E2V`. In `V2E`, each vertex's incident hyperedges are stored contiguously, and the pointer array `V2EP` records the start index of each vertex's segment in `V2E`. Similarly, `E2V` stores each hyperedge's incident vertices (pins), with `E2VP` recording the start index of each hyperedge's segment in `E2V`. While this *statically* packed data structure is well-suited for a GPU, it makes modifying the hypergraph difficult without fully rebuilding the data structure. For example, inserting a new pin into `E2V` requires shifting pins in `E2V` and updating all affected values in `E2VP`.

To address this challenge, one possible approach is to adopt iG-kway's [26] bucket-list data structure, which stores each vertex's incident edges in pre-allocated buckets to enable efficient graph modification on a GPU without rebuilding the entire CSR. However, this approach does not extend well to hypergraphs. Hypergraphs require maintaining both `V2E` and `E2V`, and replacing them with bucket-list data structures would significantly increase memory usage and limit the size of hypergraphs that can fit on a single GPU. As a result, we introduce a scalable delta-based hypergraph data structure that

efficiently supports dynamic modifications without requiring a rebuild of the original CSR structure. Our data structure consists of two components: the base hypergraph H, which stores the original incidences of vertices and hyperedges in V2E and E2V, and the delta hypergraph δH, which records the updated incidences of modified vertices and hyperedges in δV2E and δE2V. Figure 2 shows our delta-

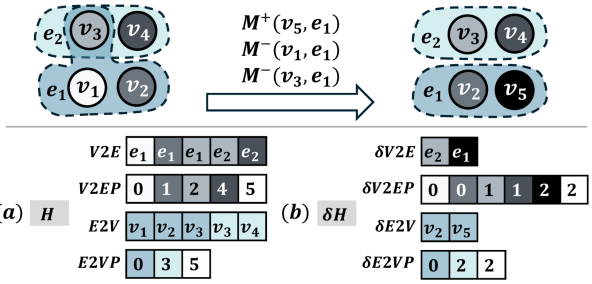


Figure 2: An example of our delta-based hypergraph data structure with modifiers $M^+_{(v_5,e_1)}$, $M^-_{(v_1,e_1)}$, and $M^-_{(v_3,e_1)}$, where (a) is the base hypergraph H and (b) is the delta hypergraph δH.

based hypergraph data structure, where (a) is the base hypergraph H, and (b) is the delta hypergraph δH, which records the updated incidences of modified vertices v_1, v_3, and v_5, as well as hyperedge e_1, after applying modifiers $M^+_{(v_5,e_1)}$, $M^-_{(v_1,e_1)}$, and $M^-_{(v_3,e_1)}$. Vertex v_3 is disconnected from e_1, and a new vertex v_5 is inserted and connected to e_1, resulting in updated incidences $\{e_2\}$ for v_3 and $\{e_1\}$ for v_5. Additionally, vertex v_1 is disconnected from its only incident hyperedge, yielding an empty incidence. The updated incidences of the modified vertices are then recorded in δV2E, with δV2EP recording the start and end indices of each vertex's updated incidence within δV2E. Similarly, the updated incidence $\{v_2, v_5\}$ of hyperedge e_1 is stored in δE2V. The array δE2VP records the start and end indices of all updated hyperedges within δE2V. For unmodified vertices and hyperedges, their start and end indices in δV2EP and δE2VP are equal, indicating that no updated incidence is recorded.

Algorithm 1: Warp-level Deletion

```
Input : M^-_(v,e), v's incident hyperedges in shared memory, smem_e
1  parallel for each thread in a GPU warp
2      lane_id ← thread index in the GPU warp
3      e_cnt ← 0
4      for j ← 0 to SHARE_SIZE/WARP_SIZE − 1 do
5          entry ← smem_e[j × WARP_SIZE + lane_id]
6          all_empty ← __all_sync(FULL, entry == ∅)
7          if all_empty > 0
8              return
9          active ← __ballot_sync(FULL, entry ≠ ∅ ∧ entry ≠ e)
10         rank ← __popc(active&((1u« lane_id) − 1u))
11         if entry ≠ ∅ ∧ entry ≠ e
12             smem_e[e_cnt + rank] ← entry
13         if lane_id == 0
14             e_cnt ← e_cnt + __popc(active)
```

4.2 Delta-based Hypergraph Construction

Since the updated incidence of modified vertices in δV2E and modified hyperedges in δE2V can be computed independently, we perform these computations concurrently by assigning them to different CUDA streams. A CUDA stream runs a sequence of GPU operations in first-in, first-out order. Using multiple streams allows independent operations to execute concurrently [8]. In the remaining sections, we focus on computing δV2E, as the same method can be applied to compute δE2V.

To compute δV2E, we assign each modified vertex v to a GPU warp (32 threads). Each warp applies the modifiers involving v sequentially to update its original incidence in V2E and writes the result to δV2E. Since both V2E and δV2E reside in high-latency global memory, we use low-latency shared memory to avoid frequent global memory accesses. Specifically, we assign the warp a shared-memory segment whose capacity exceeds the size of v's original incidence to accommodate insertions, and initialize all entries in the segment to the empty entry $\emptyset$. The warp then loads the original incidence of v into its designated shared-memory segment, applies v's associated modifiers to update its incidence in shared memory, and writes the fully updated incidence back to δV2E. This design significantly reduces memory latency, thereby improving kernel performance. To further optimize our kernel efficiency, we leverage CUDA warp-level primitives to efficiently handle deletion and insertion modifiers for each warp.

4.2.1 Hyperedge Deletion. For the warp to apply a deletion modifier $M^-_{(v,e)}$ to v's incidence in shared memory, we invoke our warp-level deletion algorithm, where all threads cooperatively identify and remove the incident hyperedge e, and shift the remaining hyperedges to ensure that v's incidence remains contiguous using efficient CUDA warp primitives. Algorithm 1 presents our warp-level deletion algorithm. The warp iterates over all entries in shared memory, and in each iteration assigns one entry to each thread to process in parallel (lines 4-5). The threads then employ the CUDA warp-level primitive `__all_sync` to cooperatively check, via fast intra-warp communication, whether all threads' assigned entries are empty (line 6). If this condition is met, it indicates that all incident hyperedges of v have been processed, and the threads terminate early (lines 7-8). Otherwise, the threads cooperatively filter out those whose assigned entries are either empty or equal to the hyperedge e using `__ballot_sync` (line 9). The remaining threads then compute their compacted indices using `__popc`, and store the content of their assigned entries to the corresponding compacted locations in shared memory (lines 10-12). Finally, the first thread updates the current hyperedge count by adding the number of remaining threads (lines 13-14).

Algorithm 2: Warp-level Insertion

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

4.2.2 Hyperedge Insertion. For the warp to apply an insertion modifier $M^+_{(v,e)}$ to v's incidence in shared memory, we invoke our warp-level insertion algorithm, where all threads cooperatively search for an empty entry in shared memory and replace it with e. Algorithm 2 presents our warp-level insertion algorithm. As in the deletion algorithm, the warp iterates over all entries in shared memory by

assigning one entry per thread (lines 3-4). All the threads then cooperatively identify the first thread whose assigned entry is empty using `__ballot_sync` and `__ffs` (lines 5-6). If such a thread exists, that thread updates its assigned entry in shared memory with e and the algorithm terminates early (lines 7-9). Otherwise, all threads move on to the next iteration to continue looking for an empty entry.

5 Incremental Hypergraph Partitioning

Once the hypergraph is modified, the existing partitioning result may become invalid due to changes in the hypergraph structure and balance condition. For example, inserting or deleting vertices can violate the balance constraint, while modifying hyperedges' incident vertices may require moving them to reduce the cut size. To refine the modified hypergraph, a straightforward approach is to apply HyperG [27] to repartition it from scratch. However, this can incur significant overhead due to redundant computations, as most of the existing partitioning remains valid after small modifications.

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

Algorithm 3: Single-pass Rebalancing

```
   Input: V2E, V2EP, δV2E, δV2EP
1  Sort vertices by partition ID
2  parallel for each vertex v assigned to a GPU warp
3      if P(v) is not overweight
4          return
5      start ← v is modified ? δV2EP[v] : V2EP[v]
6      end ← v is modified ? δV2EP[v+1] : V2EP[v+1]
7      incidence ← v is modified ? δV2E[start : end] : V2E[start : end]
8      foreach e ∈ incidence do
9          Threads cooperatively check if v is cut-critical to e and update its score
10 Segmented sort vertices by descending score using ModernGPU
11 Move top-scoring vertices from each overweight partition to P_pseudo
```

5.1 Single-pass Rebalancing

The goal of this step is to restore partition balance after the hypergraph has been modified. To do that, a common approach is to move vertices from overweight partitions to underweight ones until all partitions satisfy the balance constraint. However, concurrently moving vertices across partitions may require many iterations to achieve balance, especially when the number of partitions is large. For instance, if many vertices attempt to move to the same underweight partition, some moves must be rejected to prevent overloading that partition, and those vertices need to find other partitions in subsequent iterations. This iterative process largely underutilizes the massive parallelism available on a GPU.

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

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

Algorithm 4: Most Suitable Partition Computation

```
   Input: V2E, V2EP, δV2E, δV2EP
1  parallel for each non-adjacent vertex v assigned to a GPU warp
2      start ← v is modified? δV2EP[v] : V2EP[v]
3      end ← v is modified? δV2EP[v+1] : V2EP[v+1]
4      num_e ← end − start
5      incidence ← v is modified? δV2E[start : end] : V2E[start : end]
6      p_ms ← 0
7      Δλ_{v→p_ms} ← INT_MAX
8      foreach i ∈ [0, k−1] such that W_{p_i} + W_v ≤ W_{p_max} do
9          Δλ_{v→p_i} ← 0
10         for j ← 0 to ⌈num_e/WARP_SIZE⌉ − 1 do
11             e_idx ← j × WARP_SIZE + lane_id
12             e ← e_idx < num_e ? incidence[e_idx] : ∅
13             Δλ(e) ← (#pins of e in p_i == 0) ? 1 : 0
14             sum ← __reduce_add_sync(FULL, Δλ(e))
15             if lane_id == 0
16                 Δλ_{v→p_i} += sum
17         if Δλ_{v→p_i} < Δλ_{v→p_ms} ∧ lane_id == 0
18             Δλ_{v→p_ms} ← Δλ_{v→p_i}
19             p_ms ← p_i
```

5.2 Incremental Hypergraph Refinement

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

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

However, moving vertices in parallel can result in the incorrect selection of the most suitable partition when adjacent vertices (i.e., vertices that share the same hyperedge) are moved simultaneously [27]. To address this issue, we use a similar approach to [26] to select non-adjacent vertices from the pseudo partition and then move them in parallel. Algorithm 4 presents our most suitable partition computation algorithm, where each non-adjacent vertex v is assigned to a GPU warp to compute its most suitable partition p_{ms} and the corresponding $\Delta\lambda_{v\to p_{\text{ms}}}$. The warp first fetches v's incidence information from δV2E if v has been modified, or from V2E otherwise (lines 2-5). Next, the warp iterates over each partition p_i that allows v to move without violating the balance constraint, and computes the corresponding $\Delta\lambda_{v\to p_i}$ (line 8). To compute $\Delta\lambda_{v\to p_i}$, each thread examines one of v's incident hyperedges e to check whether e has any pins in p_i. If not, the thread sets $\Delta\lambda(e) = 1$, indicating that moving v to p_i would increase the connectivity of e by one (lines 11-13). Threads then use `__reduce_add_sync` to efficiently compute the sum of all threads' $\Delta\lambda(e)$ values across the warp and increment $\Delta\lambda_{v\to p_i}$ accordingly (lines 14-16). All threads then repeat this procedure for the next 32 hyperedges, continuing until all of v's incident hyperedges have been examined. Finally, the first thread compares $\Delta\lambda_{v\to p_i}$ with $\Delta\lambda_{v\to p_{\text{ms}}}$, and updates the most suitable partition p_{ms} to p_i if it results in a smaller increase in connectivity (lines 17-19).

After computing the most suitable partition for each non-adjacent vertex, we sort them in ascending order of $\Delta\lambda_{v\to p_{\text{ms}}}$ to prioritize moving vertices with smaller increases in connectivity. We then adopt G-kway [28]'s sequence-based strategy to determine the maximum number of vertices that can be moved without violating the balance constraint and move them in parallel. We repeat this process until all vertices in the pseudo partition have been moved to their most suitable partitions.

6 Experimental Evaluation

We evaluated the performance of iHyperG on 18 industrial circuit hypergraphs derived from the ISPD98 VLSI Circuit Benchmark Suite [1]. Since the original circuits were small (a few thousand vertices), we expanded them 100–1000 times larger with random vertex and hyperedge insertions to demonstrate the advantage of GPU parallelism. In our experiment, we applied 100 incremental iterations based on the setting of TAU Incremental Timing Contest [17] and iG-kway [26], where each iteration involved tens to hundreds of hypergraph modifiers that randomly removed/inserted vertices and hyperedges from/into the circuits.

We used HyperG [27], a state-of-the-art GPU-accelerated k-way hypergraph partitioner, as our baseline. Since HyperG did not support IHP, we rebuilt its hypergraph CSR on the GPU and repartitioned the modified circuit from that CSR. While this rebuilding strategy may not have been optimal, it represented a straightforward extension of FHP to IHP. Further optimization of this is beyond the scope of this paper. To highlight the advantages of iHyperG's incremental approach in both hypergraph modification and partitioning, we reported modification and partitioning time separately in the overall performance discussion. We implemented iHyperG and HyperG using C++17 and CUDA 12.0 and compiled them with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 16 Intel i7-11700 CPU cores at 2.50 GHz and 128 GB RAM. Our GPU was an A6000 with 48 GB of memory. In all experiments, we set the imbalance ratio ϵ to 3%. All results were averaged over 10 runs.

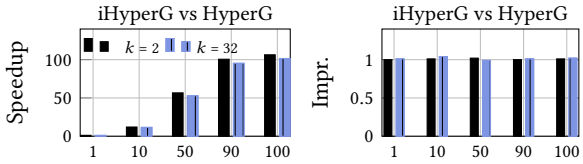


Figure 3: The speedup (left) and cut size improvement (right) of iHyperG over HyperG on Circuit05 over 100 incremental iterations, each with 25 modifiers.

6.1 Overall Performance Comparison

Table 1 compares the runtime and cut size between iHyperG and HyperG at $k = 2$ over 100 incremental iterations, each with 25 modifiers. A cut size improvement greater than one indicates that iHyperG achieves a smaller cut size than HyperG. Since HyperG rebuilds the entire CSR at each iteration, iHyperG is consistently faster in the modification stage. Instead of rebuilding, iHyperG maintains a delta-based hypergraph data structure that records the updates to modified vertices and hyperedges. Moreover, to repartition from scratch, HyperG must rebuild an additional vertex–neighbor structure. In contrast, iHyperG avoids full repartitioning and therefore does not rebuild this structure, saving significant time in the modification stage. As a result, iHyperG achieves an average speedup of 190× and up to 402× over HyperG in modification time.

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

Figure 3 shows the speedup of total partitioning time (modification and partitioning) and cut size improvement over 100 incremental iterations for Circuit05 at two extreme k values. At the first iteration, there is no significant difference in total runtime between iHyperG and HyperG since both are FHP. However, as the number of incremental iterations increases, iHyperG's runtime advantage over HyperG becomes more pronounced. The speedup of iHyperG grows roughly in proportion to the number of incremental iterations for both k values, as HyperG repartitions from scratch each iteration, whereas iHyperG refines only cut-critical vertices, saving substantial partitioning time. For the cut size, iHyperG consistently matches HyperG across all iterations. This matching in partitioning quality demonstrates that,

Table 1: Runtime and cut size comparison between iHyperG and HyperG at $k = 2$ over 100 incremental iterations, each with 25 modifiers. All times are measured in seconds. A cut size improvement above one indicates that iHyperG finds a better cut size.

Benchmark			Modification Time (s)			Partitioning Time (s)			Cut Size		
Name	# Vertices	# Edges	iHyperG	HyperG	Speedup	iHyperG	HyperG	Speedup	iHyperG	HyperG	Impr.
Circuit01	2,639,746	2,921,135	0.07	3.71	53.00×	0.67	51.01	76.13×	1,526	**1510**	0.99
Circuit02	5,076,703	5,072,388	0.10	23.46	234.60×	2.03	187.26	92.25×	**1,572**	1,591	1.01
Circuit03	3,216,039	3,808,881	0.08	11.62	145.25×	1.16	92.54	79.78×	**1,680**	1,697	1.01
Circuit04	3,273,461	3,804,547	0.08	29.13	364.13×	1.20	98.01	81.68×	**1,699**	1,708	1.00
Circuit05	5,898,849	5,717,785	0.11	11.87	107.91×	2.83	300.45	106.17×	**841**	853	1.01
Circuit06	5,817,270	6,233,993	0.11	13.03	118.45×	1.70	141.10	83.00×	1,720	**1,701**	0.99
Circuit07	5,649,026	5,918,543	0.11	10.78	98.00×	1.62	133.25	82.25×	1,748	**1,747**	1.00
Circuit08	2,001,099	1,970,143	0.07	28.13	401.86×	1.26	116.04	92.10×	**853**	868	1.02
Circuit09	4,965,854	5,664,015	0.10	10.36	103.60×	1.18	94.79	80.33×	1,800	**1,795**	1.00
Circuit10	6,179,294	6,692,566	0.11	37.03	336.64×	1.74	138.53	79.61×	**1,821**	1,825	1.00
Circuit11	5,856,441	6,760,832	0.11	10.42	94.73×	1.41	103.53	73.43×	1,802	1802	1.00
Circuit12	3,767,136	4,285,768	0.09	23.78	264.22×	1.63	129.32	79.34×	**1,810**	1,811	1.00
Circuit13	3,620,692	4,285,768	0.09	8.82	98.00×	1.16	81.03	69.85×	1,837	**1,835**	1.00
Circuit14	5,166,292	5,347,138	0.10	21.72	217.20×	1.50	123.19	82.13×	1,849	**1,848**	1.00
Circuit15	7,917,046	9,143,924	0.14	47.22	337.29×	2.15	173.71	80.80×	883	883	1.00
Circuit16	7,889,952	8,172,196	0.13	19.11	147.00×	2.16	180.19	83.42×	1,866	1,866	1.00
Circuit17	11,686,333	11,943,736	0.17	23.39	137.59×	3.70	274.99	74.32×	**1,861**	1,870	1.00
Circuit18	7,371,509	7,067,343	0.12	19.64	163.67×	2.50	235.51	94.20×	1,852	1,852	1.00
Average					**190.17×**			**82.82×**			**1.00**

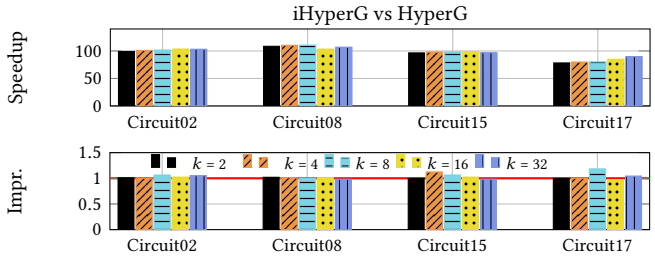


Figure 4: The speedup (top) and cut size improvement (bottom) of iHyperG over HyperG at different k values.

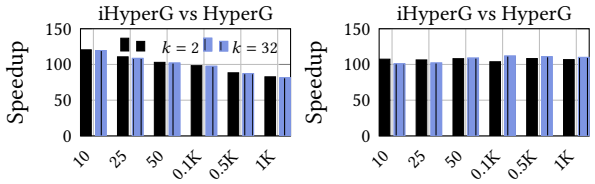


Figure 5: Speedup in hypergraph modification time (left) and hypergraph partitioning time (right) for iHyperG over HyperG on Circuit05 after 100 incremental iterations, with 10–1K modifiers per iteration.

under small modifications, incremental partitioning achieves comparable results at a fraction of the runtime, making it a more efficient alternative to full repartitioning.

6.2 Runtime and Cut Size under Varying k

Figure 4 shows the speedup of total partitioning time (modification and partitioning) and cut size improvement of iHyperG over HyperG at $k = \{2, 4, 8, 16, 32\}$ on four circuits, including the smallest (Circuit08) and the largest (Circuit17). HyperG achieves similar speedups across all k values, as both methods evaluate more partition assignments as k increases. Consequently, their runtimes grow similarly with k, and the speedup remains nearly constant.

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

6.3 Incrementality Analysis

Figure 5 shows the speedup in hypergraph modification and hypergraph partitioning time for iHyperG over HyperG on Circuit05 after 100 incremental iterations, with 10–1K modifiers per iteration. For hypergraph modification, the speedup is similar for both k values. However, it decreases as the number of modifiers grows because iHyperG must update larger delta structures with more modified vertices and hyperedges. On the other hand, HyperG rebuilds the CSR every time, so its modification time remains roughly constant even as the number of modifiers increases. For hypergraph partitioning, we do not observe a strong dependence on the number of modifiers or the value of k. This is because hyperedges typically contain many pins, so a single pin change often does not affect whether the hyperedge is cut, leaving the overall cut size unchanged. Building on this insight, our incremental refinement refines only cut-critical vertices, keeping the refinement workload relatively stable even as the number of modifiers increases.

7 Conclusion

In this paper, we present iHyperG, a GPU-parallel incremental k-way hypergraph partitioner. Experimental results show that iHyperG achieves average speedups of 190× for modification and 83× for partitioning over the state-of-the-art GPU-parallel full partitioner, HyperG [27], while maintaining comparable partitioning quality.

Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

[1] Charles J Alpert. 1998. The ISPD98 circuit benchmark suite. In *Proceedings of the 1998 international symposium on Physical design*. 80–85.

[2] Sean Baxter. 2016. moderngpu 2.0. (2016). https://github.com/moderngpu/moderngpu/wiki.

[3] Yi-Hua Chung, Shui Jiang, Wan-Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. Simpart: A simple yet effective replication-aided partitioning algorithm for logic simulation on gpu. In *European Conference on Parallel Processing*. Springer, 197–210.

[4] Dong Dai, Wei Zhang, and Yong Chen. 2017. 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*. 219–230.

[5] Lisa Durbeck and Peter Athanas. 2020. Incremental streaming graph partitioning. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[6] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1261–1274.

[7] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Scalable Shared-Memory Hypergraph Partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 16–30.

[8] Design Guide. 2013. Cuda c programming guide. *NVIDIA, July* 29, 31 (2013), 6.

[9] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[10] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[11] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*.

[12] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[13] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated static timing analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

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

[15] Zhuolun He, Yuzhe Ma, and Bei Yu. 2022. X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 52, 9 pages. doi:10.1145/3508352.3549383

[16] Zhuolun He, Yihang Zuo, Jiaxi Jiang, Haisheng Zheng, Yuzhe Ma, and Bei Yu. 2023. OpenDRC: An Efficient Open-Source Design Rule Checking Engine with Hierarchical GPU Acceleration. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC56929.2023.10247734

[17] Jin Hu, Greg Schaeffer, and Vibhor Garg. 2015. TAU 2015 contest on incremental timing analysis. In *IEEE/ACM ICCAD*. 882–889.

[18] 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).

[19] 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)*.

[20] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.

[21] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[22] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 895–902.

[23] Jiaxi Jiang, Lancheng Zou, Wenqian Zhao, Zhuolun He, Tinghuan Chen, and Bei Yu. 2024. 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* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 119, 6 pages. doi:10.1145/3649329.3657367

[24] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. 2016. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* 10 (2016), 462–476.

[25] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1997. Multilevel hypergraph partitioning: Application in VLSI domain. In *IEEE/ACM DAC*. 526–529.

[26] 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 *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[27] 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 *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. 1031–1040.

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

[29] Wan Luan Lee, Dian-Lun Lin, Shui Jiang, Cheng-Hsiang Chiu, Yibo Lin, Bei Yu, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner using Task Graph Parallelism. *ACM Transactions on Design Automation of Electronic Systems* (2025).

[30] Thomas Lengauer. 2012. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media.

[31] Peiyu Liao, Yuxuan Zhao, Dawei Guo, Yibo Lin, and Bei Yu. 2024. 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* 43, 6 (2024), 1624–1637. doi:10.1109/TCAD.2023.3347293

[32] 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 *ACM International Conference on Parallel Processing (ICPP)*.

[33] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*.

[34] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 71, 6 pages. doi:10.1145/3649329.3655983

[35] Shiju Lin, Jinwei Liu, Evangeline FY Young, and Martin DF Wong. 2022. Gamer: Gpu-accelerated maze routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2022), 583–593.

[36] Siting Liu, Yuan Pu, Peiyu Liao, Hongzhong Wu, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, and Bei Yu. 2022. FastGR: Global Routing on CPU–GPU With Heterogeneous Task Graph Scheduler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 7 (2022), 2317–2330.

[37] Tianji Liu, Lei Chen, Xing Li, Mingxuan Yuan, and Evangeline F.Y. Young. 2024. FineMap: A Fine-grained GPU-parallel LUT Mapping Engine. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 392–397. doi:10.1109/ASP-DAC58780.2024.10473941

[38] Yuxi Liu, Rong Ye, Feng Yuan, Rakesh Kumar, and Qiang Xu. 2012. On logic synthesis for timing speculation. In *Proceedings of the International Conference on Computer-Aided Design*. 591–596.

[39] Chedi Morchdi, Cheng-Hsiang Chiu, Wan-Luan Lee, Tsung-Wei Huang, and Yi Zhou. 2025. Enhancing Graph Partitioning with Reinforcement Learning-based Initialization. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[40] Chao-Wei Ou and Sanjay Ranka. 1997. Parallel incremental graph partitioning. *IEEE transactions on Parallel and Distributed Systems* 8, 8 (1997), 884–896.

[41] Aditya Das Sarma, Shui Jiang, Wan Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 266–272. doi:10.1109/ASP-DAC66049.2026.11420570

[42] Aditya Das Sarma, Wan-Luan Lee, Shui Jiang, Boyang Zhang, and Tsung-Wei Huang. 2026. A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation. In *ACM/IEEE Design Automation Conference (DAC)*.

[43] Jie Tong, Zhengxiong Li, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[44] Zhenlin Wu, Haosong Zhao, Hongyuan Liu, Wujie Wen, and Jiajia Li. 2025. gHyPart: GPU-friendly End-to-End Hypergraph Partitioner. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–25.

[45] Evangeline F.Y. Young. 2023. GPU Acceleration in Physical Synthesis. In *Proceedings of the 2023 International Symposium on Physical Design* (Virtual Event, USA) *(ISPD '23)*. Association for Computing Machinery, New York, NY, USA, 167. doi:10.1145/3569052.3578912

[46] Ziyang Yu, Guojin Chen, Yuzhe Ma, and Bei Yu. 2023. A GPU-Enabled Level-Set Method for Mask Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2023), 594–605. doi:10.1109/TCAD.2022.3175939

[47] 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 *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.