HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner

Wan-Luan Lee
University of Wisconsin at Madison
Madison, Wisconsin, USA

Dian-Lun Lin
University of Wisconsin at Madison
Madison, Wisconsin, USA

Cheng-Hsiang Chiu
University of Wisconsin at Madison
Madison, Wisconsin, USA

Ulf Schlichtmann
Technical University of Munich
Munich, Germany

Tsung-Wei Huang
University of Wisconsin at Madison
Madison, Wisconsin, USA

Abstract

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

ACM Reference Format:
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 *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3658617.3697551

1 Introduction

Hypergraph partitioning plays a critical role in various stages of circuit design, including placement, routing, timing analysis, and logic simulation. For instance, hypergraph partitioning helps optimize component placement on a chip by dividing the circuit into smaller, more manageable blocks while minimizing interconnections among them [70]. Given that hypergraph partitioning is NP-hard [5, 24], many heuristics have been developed [3, 6, 71]. Among these heuristics, *multilevel partitioning* stands out as the most popular for large-scale circuit graphs due to its high-quality partitioning results and fast runtime [5, 19, 24, 64, 83]. A typical multilevel hypergraph partitioner iteratively coarsens the original hypergraph into a smaller representation. When the hypergraph becomes small enough, the partitioner uses a fast algorithm to generate an initial balanced partition. Finally, the partitioner iteratively restores the graph to the previous level, followed by a refinement algorithm to improve the partition solution. Among all stages, coarsening and refinement stages are the most time-consuming and account for 90% of the total runtime [23].

As the circuit size continues to grow, multilevel hypergraph partitioning becomes increasingly time-consuming. For example, the sequential hypergraph partitioner hmetis can take four minutes to partition a five million-gate circuit [64]. Since hypergraph partitioning can be performed multiple times during a CAD algorithm (e.g., incremental timing [40], RTL simulation [80]), the cumulative partitioning time can extend to several hours. To reduce the long runtime, existing partitioners [24, 83] have utilized multi-core CPUs to parallelize the partitioning. Among many parallel graph partitioners, Mt-KaHyPar [24] is the state-of-the-art CPU-based parallel hypergraph partitioner designed to parallelize the sequential k-way Fiduccia-Mattheyses algorithm [22]. Despite some runtime improvements, the speedup typically plateaus at 8–16 CPU threads [24]. On the other hand, modern GPUs provide massive amount of parallelism and higher memory bandwidth than CPUs, offering a new opportunity to accelerate hypergraph partitioning.

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

Recent research, such as G-kway [69] and GKSG [23], has investigated the use of GPU to accelerate *non-hypergraph* partitioning (i.e., exactly two vertices per edge). To apply G-kway or GKSG to hypergraph partitioning, one potential solution is to transform a hypergraph into a non-hypergraph [21]. However, this transformation often results in poor partitioning quality, as the transformed non-hypergraph fails to accurately represent the original hypergraph [60, 64]. Another possible solution is to extend G-kway's non-hypergraph partitioning algorithm to hypergraphs. However, due to the inherent differences between nonhypergraphs and hypergraphs, this approach can result in extremely low parallelism. For example, G-kway identifies an independent set of vertices to refine in parallel. Since a hyperedge can easily link to many vertices, the size of an independent set in a hypergraph is typically small. This

ASPDAC '25, January 20–23, 2025, Tokyo, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0635-6/25/01.
https://doi.org/10.1145/3658617.3697551

situation becomes even more challenging with modern circuits, as a net typically connects to numerous pins. *Given these challenges and the importance of a fast hypergraph partitioner, there is a need for a new GPU-accelerated hypergraph partitoning algorithm.*

Consequently, we present *HyperG*, a GPU-accelerated hypergraph partitioning algorithm. While the previous work [12] has attempted to accelerate the uncoarsening stage using GPUs, to the best of our knowledge, HyperG is among the earliest attempts to parallelize both coarsening and uncoarsening stages on a GPU. The three key contributions of HyperG are summarized below:

- We introduce a balanced group coarsening algorithm that groups many vertices into balanced subgroups to ensure high partitioning quality at later initial partitioning stage.
- We introduce a sequence-based refinement algorithm that simultaneously identifies and moves the best subsequence of vertices to largely improve the partition solution.
- We develop our GPU kernel using modern CUDA warp-level primitives to achieve fine-grained synchronization and efficient communication during both the coarsening and refinement stages.

We evaluate the performance of HyperG on industrial circuit graphs and compared our results with two state-of-the-art hypergraph partitioners, sequentialy partitioner hmetis [64] and CPU-based parallel partitioner Mt-KaHyPar [24]. On average, experimental results show that HyperG outperforms hmetis and 16-threaded Mt-KaHyPar by 133× and 4.1× faster, respectively, with 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 the vertex set V representing multi-vertex relationships. The vertices that belong to a hyperedge are referred to that hyperedge's pins. We donate the size of e as $|e|$, which is equal to the number of pins belonging to e. For a vertex v, we donate the weight of $v \in V$ by W_v, while for a hyperedge $e \in E$, we donate the weight of e by W_e. Vertices u and v are neighbors, if there exists a hyperedge $e \in E$ such that $u \in e$ and $v \in e$.

Given an integer k, the goal of the hypergraph partitioning problem is to partition V into k disjoint subsets $P_1, P_2, \ldots, P_k$ of approximately equal sizes, while minimizing the cut size. The cut size is a commonly used metric to measure the interconnection among partitions and is defined as the sum of the weights of all cut hyperedges. A cut hyperedge is a hyperedge that contains pins belonging to more than one partition. For a vertex u, we define $P(u) = i$ if $v \in P_i$. The weight of partition P_i is define as $W_{P_i} = \sum_{v \in P_i} W_v$. To ensure that each partition has roughly equal sizes, a balance constraint is imposed, limiting the maximum size of each partition P_i as $W_{P_i} \le (1 + \epsilon)\frac{\sum_{v \in V} W_v}{k}$, where $0 < \epsilon \ll 1$ and ϵ is the imbalance ratio given by applications.

3 GPU Multilevel Hypergraph Partitioner

Following the multilevel heuristic, HyperG consists of three main stages: *coarsening*, *initial partitioning*, and *uncoarsening*. Figure 1 shows an overview of HyperG.

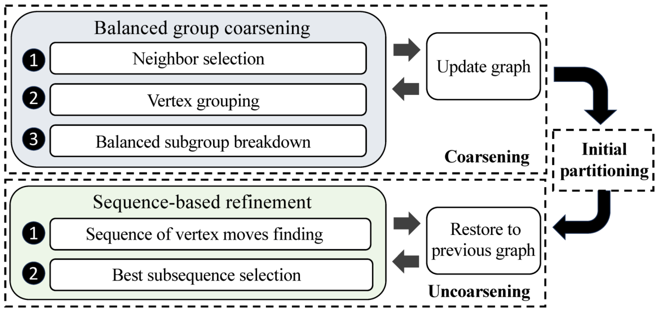
Balanced group coarsening
1 Neighbor selection
2 Vertex grouping
3 Balanced subgroup breakdown
Update graph
Coarsening
Initial partitioning
Sequence-based refinement
1 Sequence of vertex moves finding
2 Best subsequence selection
Restore to previous graph
Uncoarsening

Figure 1: Overview of HyperG that consists of three main stages: coarsening, initial partitioning, and uncoarsening.

- *Coarsening.* The goal is to coarsen the hypergraph into a smaller representation level by level while preserving the original hypergraph's structure. The coarsening process continues until the hypergraph has fewer than $160 \times k$ vertices or until less than 95% of the vertices can be coarsened in the previous level. We develop a *balanced group coarsening* algorithm that can coarsen many vertices simultaneously while ensuring that the coarsened vertices are balanced in size. This balance in vertex sizes is crucial for achieving a balanced initial partition in the initial partitioning stage.
- *Initial partitioning.* The goal is to create an initial partition from the coarsest hypergraph. We utilize the CPU-based parallel hypergraph partitioner Mt-KaHyPar [24] for the initial partitioning. Since the coarsest hypergraph is much smaller than the original hypergraph, the initial partitioning stage is very fast and does not benefit much from GPU parallelism.
- *Uncoarsening.* The goal is to iteratively restore the coarsened hypergraph back to the previous level and refine the partitioning result by moving vertices among partitions (i.e., refinement). The uncoarsening process continues until the hypergraph size is the same as the original hypergraph. We develop an efficient *sequence-based refinement* algorithm that finds the best subsequence of vertices to move in parallel, significantly improving the cut size while reducing the refinement time.

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

3.1 Balanced Group Coarsening

State-of-the-art CPU-parallel coarsening methods adopt rating function coarsening [1, 4, 24]. Each vertex initially starts in its own group. Each thread then visits a vertex, finds the neighbor with the highest score, and joins the vertex to that neighbor's group. Finally, all vertices in the same group will coarsen into a coarsened vertex. To prevent a group from becoming too large and causing an imbalanced initial partition at a later stage, Mt-KaHyPar imposes a maximum size on each group and prevents vertices from joining a group that exceeds the maximum size. Consequently, frequent checks (i.e., synchronization) are required to ensure that vertices do not join oversized groups. Such synchronization can introduce significant overhead for GPU, as each GPU thread involves frequent waiting for another GPU thread to update the correct group size, which reduces the parallel efficiency of GPU.

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

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

3.1.1 Neighbor selection. We develop an efficient GPU kernel using a highly optimized CUDA warp-level primitive, *__reduce_max_sync*, to find the neighbor with the highest score. Inspired by [24], we define the rating function for each vertex u and its neighbor v as follows:

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

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

3.1.2 Vertex grouping. After selecting the neighbor with the highest score for each vertex, we perform the vertex grouping algorithm to join vertices into groups in parallel and coarsen all vertices in the same group together. Our vertex grouping algorithm is inspired by [88], where each vertex's group ID is iteratively updated to a larger value until all vertices with connected selected neighbors share the same group ID. However, after grouping, some groups are significantly larger than others, making it difficult to achieve a balanced partition at the later initial partitioning stage. One solution is to randomly divide each large group into smaller subgroups. However, this approach can lead to poor partition quality because vertices that are far apart may end up in the same subgroup. Coarsening these distant vertices together can distort the original graph structure. Figure 2 (a) shows an example of breaking down a large group into subgroups of size two randomly. This method places two nonadjacent vertices, v_1 and v_2, into the same subgroup, which distorts the original graph structure by coarsening them together.

To address these issues, we create a group combination for each vertex, ensuring that vertices closer to each other have closer group combinations. We then divide vertices with closer group combinations into the same subgroups. In our algorithm, each vertex's group combination is an eight-byte data type, where the first four bytes store the vertex's group ID, and the last four bytes indicate the iteration during which this vertex joins the group. This setup allows us to update the group combination with a single *atomicMax* operation. Specifically, vertices closer to the group leader (i.e., the vertex with the largest vertex ID in the group) will join the group leader's group faster and adopt the smaller group combinations, while more distant vertices will join the group later and have larger group combinations. Figure 2 (b) shows an example of breaking down a large group into subgroups of size two by group combinations. This method places adjacent vertices into the same subgroups, which better preserves the original graph structure than method (a).

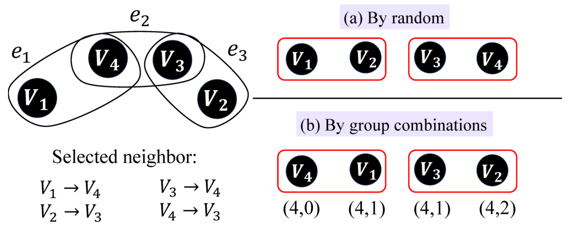

Figure 2: Examples of two methods to break down a large group into subgroups of size two. Vertices in the same box are placed into the same subgroup. $V_i \to V_j$ indicates V_i selects to join V_j's group. In (b), $(n1, n2)$ represents a vertex's group combination, where $n1$ indicates the group ID and $n2$ is the iteration number.

Algorithm 1 presents our vertex grouping algorithm. We use an array *group_comb* to store each vertex's group combination. Initially, each vertex is in its own group, and we initialize its corresponding value in *group_comb* such that the first four bytes equal this vertex's vertex ID and the last four bytes are set to 0. We utilize a boolean variable *joining* to track if there are any vertices still joining other groups. While *joining* is true, we assign each vertex to a GPU thread. Each thread is responsible for updating the group combinations of its assigned vertex and its selected neighbor by first comparing their group IDs and finding the larger one (lines 10 or 14). The thread then creates a new group combination using the larger group ID and the iteration number (lines 11 or 15) and uses *atomicMax* to update the group combination of either its vertex or its selected neighbor, whichever is smaller (lines 12 or 16). Since multiple threads may attempt to update the same vertex's group combination concurrently, we use *atomicMax* to ensure these updates are performed atomically. Finally, threads that update the group combination sets *joining* to true (lines 13 or 17), indicating that there are still vertices joining other groups and that threads need to continue updating group

Algorithm 1: Vertex grouping

```
$it \leftarrow 1$
$joining \leftarrow$ true
while ($joining$)
    $joining \leftarrow$ false
    parallel for each GPU thread
        $gid \leftarrow$ GPU thread's global ID
        $nbr \leftarrow selected\_neighbor[gid]$
        $nbr\_group\_ID \leftarrow$ get_group_ID($group\_comb[nbr]$)
        $cur\_group\_ID \leftarrow$ get_group_ID($group\_comb[gid]$)
        if $cur\_group\_ID > nbr\_group\_ID$ then
            $max\_group\_comb \leftarrow$ get_group_comb($cur\_group\_ID, it$)
            atomicMax($\&group\_comb[nbr], max\_group\_comb$)
            $joining \leftarrow$ true
        else if $cur\_group\_ID < nbr\_group\_ID$ then
            $max\_group\_comb \leftarrow$ get_group_comb($nbr\_group\_ID, it$)
            atomicMax($\&group\_comb[gid], max\_group\_comb$)
            $joining \leftarrow$ true
    $it$++
```

combinations. Once all vertices have finished joining groups and group combinations no longer change, all vertices in the same group will have group IDs equal to the vertex ID of their group leader.

Algorithm 2: Balanced subgroup breakdown

```
parallel for each group
    $gid \leftarrow$ GPU thread's global ID
    $i^{th} \leftarrow$ the $i^{th}$ vertex in the group   ▷ position within the group
    $sub\_group\_start \leftarrow$ floor($i^{th}$ / $s$)
    $sub\_group\_id \leftarrow v\_id[sub\_group\_start \times s]$
    $group\_id[gid] \leftarrow sub\_group\_id$
```

3.1.3 *Balanced subgroup breakdown.*

After computing the group combinations, we sort the vertices in ascending order based on their group combinations, so that vertices belonging to the same group are placed together. Within each group, vertices are ordered according to their distance from the group leader (i.e., vertices farther from the group leader have larger group IDs). We then divide vertices into subgroups based on their position within the group, with each subgroup having a maximum size of s. We maintain an array v_id to record each vertex's vertex ID in the same order as they appear in $group\_comb$. Algorithm 2 shows our balanced subgroup breakdown algorithm. For each group, we assign each vertex in the group to a GPU thread. The thread first identifies the index of its vertex within the group (line 3). The thread then computes the subgroup for the vertex by dividing the index by s (lines 4). To assign the subgroup a new group ID and ensure that no two subgroups share the same ID, the thread locates the first vertex within its subgroup and uses this vertex's vertex ID as the subgroup ID (line 5). Finally, the thread updates the group ID accordingly (line 6).

3.2 Sequence-based Refinement

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

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

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

- The first term $\delta(num\_pins(e, P_{cur}) = 1)$ checks if e has only one pin left in P_{cur}. If the condition is true, then moving u from P_{cur} will result in a positive contribution to the gain, as it reduces the number of partitions e spans.
- The second term $\delta(num\_pins(e, P_{dst}) = 0)$ checks if e has no pins in P_{dst}. If the condition is true, then moving u to P_{dst} will result in a negative contribution to the gain, as it increases the number of partitions e spans.

Parallel refinement can make each vertex's gain inconsistent due to the concurrent movement of adjacent vertices [69]. To ensure correct gains, G-kway [69] identifies an independent set of vertices to move in parallel. However, such a strategy largely reduces the available parallelism for hypergraphs. Since a hyperedge can connect to many vertices, the size of an independent set in a hypergraph is often small. To address this problem, we propose a sequence-based refinement algorithm. We first finds a sequence of vertex moves with positive gain in the descending order. Next, we update the gain of each vertex move in the sequence by assuming previous vertex moves are already applied. This strategy allows us to update the gain of a vertex move based on its neighbors' latest partition locations, eliminating the need for an independent set. Since a vertex move may have negative gains after updating, we accumulate the gains to identify the best subsequence of vertex moves that yields the largest gain while maintaining balanced partitions. Our sequence-based refinement algorithm consists of two steps: *Sequence of vertex moves finding* and *Best subsequence selection.*

3.2.1 *Sequence of vertex moves finding.*

In this step, our goal is to find vertices with positive gains and store them in a sequence of vertex moves *seq_vertex_moves*. To avoid redundant gain computations, we record the adjacent partitions of each vertex u (i.e., partitions where u has neighbors) using a 64-bit data type adj_par. In adj_par, each bit represents a partition and its value is either 1 or 0. Specifically, if P_i is adjacent to u, the value of the i^{th} bit in u's adj_par is 1; otherwise, it is set to 0. This information is updated as vertices are moved. We only calculate gains for the adjacent partitions of u since moving a vertex to a non-adjacent partition will not result in a positive gain. This strategy largely reduces unnecessary gain calculations.

Algorithm 3: Sequence of vertex moves finding

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

To find vertices with positive gains, we assign each GPU warp 32 consecutive vertices. All threads in a warp calculate the gain of a vertex u for one of its adjacent partitions at a time. Since each vertex can have a different number of adjacent partitions and hyperedges, the time for computing a vertex's gain is different. In our kernel, if a warp finishes processing a vertex quickly, it can immediately proceed to the next one. This approach increases warp occupancy, ensuring enough active warps to keep the GPU cores busy. Algorithm 3 shows our sequence of vertex moves finding algorithm. In each warp, all threads first fetch 32 consecutive vertices (lines 4-5), and they calculate the gain of a vertex v to one of its adjacent partitions P at a time. Specifically, each thread fetches one of u's hyperedge e and calculates the gain for e (lines 11-12). We use *__reduce_add_sync* to efficiently sum up the gain of each e computed by each thread (line 13). After obtaining the total gain, the first thread in the warp (i.e., lane ID equals 0) checks if the total gain is greater than the current maximal gain (line 14). If it is true, the thread updates the current maximal gain and the associated partition (lines 15-16). Then, all threads continue processing v's next adjacent partition. Once threads have processed all of v's adjacent partitions, the first thread checks if v has a maximal gain greater than zero (lines 18). If it does, the thread atomically increments a variable indicating the current sequence size *seq_size* to get a position in *seq_vertex_moves* and inserts the vertex move into the sequence (lines 19-20).

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

Algorithm 4: Gain updating

```
assign a $m_u^{P_{cur}, P_{dst}}$ to a GPU warp
parallel for each GPU warp
    lane_id ← GPU thread's ID within a warp
    u_move_order ← move_order[u]
    gain ← 0
    foreach u's hyperedge e
        pin_id ← lane_id$^{th}$ pin in e
        pin_move_order ← move_order[pin_id]
        pin_par ← pin_move_order < u_move_order ?
          seq_vertex_moves [pin_move_order].$dst$ : seq_vertex_moves
          [pin_move_order].$cur$
        num_pins_cur ← number of pin_pars are $m_u^{P_{cur}, P_{dst}}.cur$
        num_pins_dst ← number of pin_pars are $m_u^{P_{cur}, P_{dst}}.dst$
        if lane_id == 0 && num_pins_dst == 0 then
            gain − = $W_e$
        if lane_id == 0 && num_pins_cur == 1 then
            gain + = $W_e$
        __syncwarp()
    if lane_id == 0 then
        seq_vertex_moves[u_move_order] .gain ← gain
```

3.2.2 *Best subsequence selection.*

The goal of this step is to select the best subsequence of vertex moves to move. We first update the gain of each vertex moves by assuming previous vertex moves are all applied. The gain of a vertex move is updated by computing its neighbors' current partitions based on their order in the sequence of vertex moves. To allow each thread to quickly access a vertex move's order without searching the entire sequence, we maintain a *move_order* array of length $|V|$, where the index of vertex u records u's order in the sequence. Algorithm 4 presents our parallel gain updating algorithm. We assign each vertex move $m_u^{P_{cur}, P_{dst}}$ to a GPU warp, where all threads in the warp update the gain of $gain(u, P_{dst})$, by computing the gain of each hyperedge e of u at a time. Each thread first fetches one of e's pin *pin* and its order *pin_move_order* from *move_order* to a thread-local variable (lines 7-8). Each thread then finds *pin*'s current partition *pin_par* by comparing the order of *pin* and u (lines 9). If *pin*'s order is less than u's, meaning *pin* moves before u, then by the time we are moving u, *pin* should already be in its destination partition. Conversely, if *pin*'s order is greater than u's, meaning *pin* moves after u, then by the time we are moving u, *pin* should still be in its current partition. Based on the order of *pin*, each thread determines its *pin*'s current partition and counts the number of pins in $m_u^{P_{cur}, P_{dst}}$'s current partition P_{cur} (lines 10). To efficiently calculate the number of pins in the P_{cur}, we use *__ballot_sync* to identify threads whose *pin_par* matches P_{cur} and *__popc* to count those threads. The number of pins in the destination partition is computed similarly (lines 11). Finally, the first thread in the warp checks if e has no pins in P_{dst}. If this condition is met, e results in a negative gain, and the thread decrements the gain by W_e (lines 12-13). Additionally, the thread checks if e has only one pin left in P_{cur}. If this condition is met, e results in a positive gain, and the thread increments the gain by W_e (lines 14-15).

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

Next, we use the same strategy to compute the partition balance result. We maintain k different del_wgt_i arrays for k partitions, each with a size equal to the number of vertex moves in the sequence. Each element in del_wgt_i records the weight change of the i^{th} partition after applying a vertex move. Specifically, the value in $del\_wgt_1[0]$ records the first partition's weight change after applying the first vertex move. We then use a GPU scan on each del_wgt_i to get the total partition weight change for each partition. After scans, the j^{th} element in del_wgt_i stores the total partition weight change for partition i from applying the first to the j^{th} vertex moves. Based on the total partition weight changes, we can compute if the partition will be balanced after applying vertex moves in the subsequence. Using the accumulated gains *accum_gains* and the result of each del_wgt_i, we can find the longest subsequence where the total gain is maximized and the resulting partition remains balanced. We then apply all vertex moves in this subsequence in parallel.

4 Experimental Evaluation

We evaluated the performance of HyperG on 18 industrial circuit graphs derived from the ISPD98 VLSI Circuit Benchmark Suite [2]. Since the original graphs are small (a few thousand vertices), we expanded the circuit graphs 100–1000 times larger with random vertex and edge insertions to demonstrate the advantage of GPU parallelism. We implemented HyperG using C++17 and CUDA 12.0 and compiled it with nvcc on a host compiler of GCC-8 with -O3 enabled. We ran experiments on a 64-bit Linux machine with 16 Intel i7-11700 CPU cores at 2.50 GHz and 128 GB RAM. Our GPU is A6000 with 48 GB global memory.

4.1 Baselines

We consider two state-of-the-art hypergraph partitioners as our baseline, hmetis v1.5 [64] (sequential) and Mt-KaHyPar [24] (multithreaded). For all experiments, we set the imbalance ratio to 3%. In the coarsening stage, we set the maximum size of each subgroup to four and terminate the coarsening algorithm when the number of vertices drops below $160 \times k$ or less than 95% of the vertices can be coarsened in the previous coarsening level. We use the default settings for both hmetis and Mt-KaHyPar.

4.2 Overall Performance Comparison

Table 1 compares the overall runtime and cut size results among hmetis, Mt-KaHyPar, and HyperG at $k = 2$. On our machine, Mt-KaHyPar saturates at about 10–16 threads. Hence, we report its results under 16 threads. In terms of runtime, HyperG outperforms hmetis and Mt-KaHyPar across nearly all circuit graphs, with an average speedup of 133× and 4.1×, respectively. The largest speedups

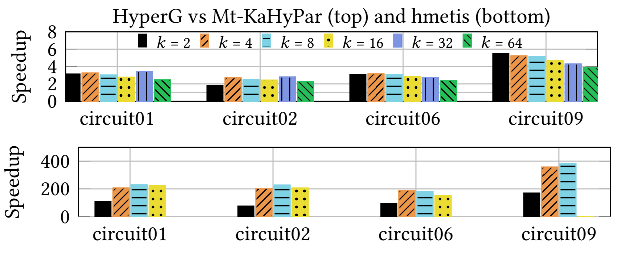

Figure 3: The speedup of HyperG over Mt-KaHyPar (top) and hmetis (bottom) at different k. In cases where hmetis fails to partition the circuit graph, the results are left blank.

we observe are 177× over hmetis and 6× over Mt-KaHyPar. For the smallest graph (circuit08), the runtime of HyperG is slightly slower than Mt-KaHyPar (0.9×), which is due to the limited data parallelism exhibited by this graph. For the largest graph (circuit17), hmetis failed to finish due to memory error. Regarding cut size, HyperG always achieves comparable values with both hmetis and My-KaHyPar.

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

4.3 Runtime Analysis

Figure 3 shows the speedup of HyperG over Mt-KaHypar (16 threads) and hmetis at $k = \{2, 4, 8, 16, 32, 64\}$ on four circuit graphs, circuit01, circuit02, circuit06, and circuit09. We chose these four circuit graphs because hmetis can finish the partitioning in most k during our settings.

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

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

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

Table 1: Overall comparison of runtime (second) and cut size among hmetis (sequential), Mt-KaHyPar (16 threads), and HyperG at $k = 2$. The last two columns show the speedup of HyperG over hmetis and Mt-KaHyPar. Best cut sizes are in bold.

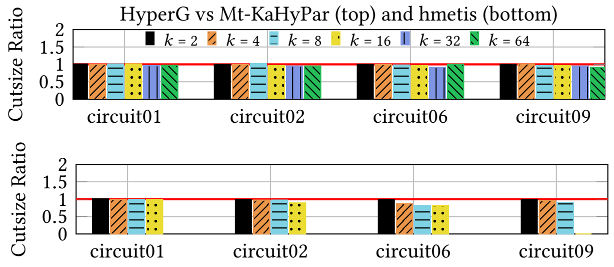

Figure 4: The cut size ratio of HyperG over Mt-KaHyPar (top) and hmetis (bottom) at different k. In cases where hmetis fails to partition the circuit graph, the results are left blank.

4.4 Cut Size Analysis

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

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

4.5 Scalability Analysis

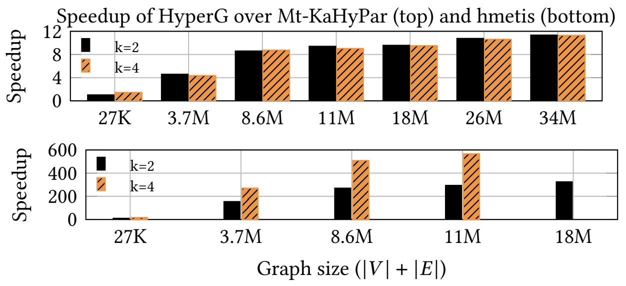

Figure 5: The speedup of HyperG over Mt-KaHyPar (top) and hmetis (bottom) for varying circuit graph sizes modified from ibm01 at $k = 2$ and $k = 4$. hmetis fails to partition the circuit graph with size larger than 18M.

Figure 5 shows the speedup of HyperG over Mt-KaHyPar and hmetis when partitioning circuit graphs of varying sizes at $k = 2$ and $k = 4$. We choose only these two k values because hmetis fails to partition the graphs at larger k values. We generate different circuit graph sizes ($|V| + |E|$) by randomly inserting vertices and edges to enlarge ibm01 [2] from 27K to 34M. When the graph size is small, the speedup is subtle. For example, at 27K, the speedup of HyperG over Mt-KaHyPar and hmetis is only 1.2× and 10×, respectively. However, as the graph size goes beyond 3M vertices, HyperG shows a significant performance advantage over the CPU-based parallel Mt-KaHyPar and sequential hmetis. Moreover, hmetis fails to partition graphs beyond 18M. For example, for the largest graph (34M), HyperG achieves an 11× speedup over Mt-KaHyPar, while hmetis fails to partition the graph. The speedup of HyperG continues to increase as the graph turns larger, showing the advantage of GPU acceleration for large-scale graph partitioning.

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

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

4.6 Comparison with Graph Partitioners

Table 2 compares the cut size and runtime results between two GPU-accelerated partitioners, HyperG (hypergraph) and G-kway (non-hypergraph) [69] at $k = 2$. Since G-kway is a non-hypergraph partitioner, it requires an extra transformation of hypergraphs into non-hypergraphs. To transform a hypergraph to a non-hypergraph, we use a classical clique-expansion method where each hyperedge is replaced with a clique [21, 64]. In all benchmarks, HyperG consistently produces better cut sizes than G-kway. This is because HyperG can work directly on the hypergraph, preserving the original multi-vertex relationships and effectively minimizing the cut size. However, G-kway introduces many more edges by transforming the hypergraph to a non-hypergraph, which leads to an increased edge count and a loss of the original hypergraph structure.

5 Conclusion

In this paper, we have introduced HyperG, a GPU-accelerated hypergraph partitioner to achieve significant runtime improvement that was previously out of reach with CPU. HyperG introduces an innovative balanced group coarsening and a sequence-based refinement algorithm to accelerate both the coarsening and uncoarsening stages. Experimental results have shown promising performance of HyperG over state-of-the-art CPU-parallel hypergraph partitioners on large industrial circuits. Inspired by the success of GPU computing in graph processing [7–11, 13–18, 20, 25–59, 61–63, 65–69, 72–82, 84–87, 89–91], we plan to enhance HyperG using CUDA Task Graph parallelism.

Acknowledgement

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

References

[1] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a direct k-way hypergraph partitioning algorithm. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 28–42.

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

[3] Author(s). 2023. An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design. In *IEEE/ACM ICCAD*.

[4] Umit V Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on parallel and distributed systems* 10, 7 (1999), 673–693.

[5] Ümit V Çatalyürek and Cevdet Aykanat. 2011. PaToH (Partitioning Tool for Hypergraphs).

[6] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* 55, 12, Article 253 (mar 2023), 38 pages.

[7] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. 2024. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[8] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC*.

[9] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[10] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[11] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM ICPP*. 565–575.

[12] Lin Cheng, Hyunsu Cho, and Peter Yoon. 2015. An accelerated procedure for hypergraph coarsening on the GPU. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[13] Cheng-Hsiang Chiu and Tsung-Wei Huang. [n. d.]. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[14] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*.

[16] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*.

[17] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

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

[19] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10–pp.

[20] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.

[21] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 3558–3565.

[22] Charles M. Fiduccia and Robert M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *ACM/IEEE DAC*. 175–181.

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

[24] 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.

[25] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*.

[26] 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).

[27] 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)*.

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

[29] 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)*.

[30] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[31] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs.

In *IEEE/ACM Design Automation Conference (DAC)*.
[32] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[33] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).
[34] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.
[35] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.
[36] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.
[37] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*.
[38] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[39] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
[40] 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).
[41] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[42] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*.
[43] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*.
[44] 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)*.
[45] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*.
[46] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.
[47] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).
[48] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)* (2021).
[49] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).
[50] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).
[51] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*.
[52] Tsung-Wei Huang, Hong-Yan Su, and Tsung-Yi Ho. 2011. Progressive network-flow based power-aware broadcast addressing for pin-constrained digital microfluidic biochips. In *ACM/IEEE Design Automation Conference (DAC)*. 741–746.
[53] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[54] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2016).
[55] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*.
[56] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD*.
[57] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Route: An Ultra-Fast Incremental Maze Routing Algorithm. In *ACM System Level Interconnect Prediction Workshop (SLIP)*. 1–8.
[58] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD*.
[59] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*.
[60] Edmund Ihler, Dorothea Wagner, and Frank Wagner. 1993. Modeling hypergraphs by graphs with the same mincut properties. *Inform. Process. Lett.* 45, 4 (1993), 171–175.
[61] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[62] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*.
[63] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM ICPP*. 388–399.
[64] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1997. Multilevel hypergraph partitioning: Application in VLSI domain. In *IEEE/ACM DAC*. 526–529.
[65] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.
[66] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[67] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.
[68] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[69] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*.
[70] Thomas Lengauer. 2012. *Combinatorial algorithms for integrated circuit layout.* Springer Science & Business Media.
[71] Rongjian Liang, Anthony Agnesina, and Haoxing Ren. 2024. MedPart: A Multi-Level Evolutionary Differentiable Hypergraph Partitioner. In *Proceedings of the 2024 International Symposium on Physical Design*. 3–11.
[72] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*.
[73] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[74] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.
[75] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*.
[76] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[77] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.
[78] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).
[79] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[80] 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)*.
[81] 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)*.
[82] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.
[83] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. Bipart: a parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming. 161–174.

[84] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[85] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.

[86] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. 2024. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[87] Sheng-Han Yeh, Jia-Wen Chang, Tsung-Wei Huang, Shang-Tsung Yu, and Tsung-Yi Ho. 2014. Voltage-Aware Chip-Level Design for Reliability-Driven Pin-Constrained EWOD Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 33, 9 (2014), 1302–1315.

[88] Kensaku Yonehara and Kunio Aizawa. 2015. A line-based connected component labeling algorithm using GPUs. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 341–345.

[89] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[90] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[91] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.