InstantGR: Scalable GPU Parallelization for 3-D Global Routing

Liang Xiao, Shiju Lin, Jinwei Liu, Qinkai Duan, Tsung-Yi Ho, *Fellow, IEEE*, and Evangeline F. Y. Young, *Fellow, IEEE*

***Abstract*—Global routing plays a crucial role in electronic design automation (EDA), serving not only as a means of optimizing routing but also as a tool for estimating routability in earlier stages, such as logic synthesis and physical planning. However, these scenarios often require global routing on unpartitioned large designs, posing unique challenges in scalability, both in terms of runtime and design size. To tackle this issue, this article introduces useful techniques for parallelizing large-scale global routing that can significantly increase parallelism and thus reduce runtime. We also propose a new flexible layer transition technique to increase the flexibility and routing quality of directed acyclic graph (DAG) routing. Building upon these techniques, we have developed an open-source GPU-based global router that achieves state-of-the-art results in the latest ISPD'24 Contest benchmarks, thereby showcasing the effectiveness of our methods.**

***Index Terms*—Electronic design automation (EDA), global routing, graphics processing unit (GPU) acceleration, physical design.**

I. Introduction

ROUTING is a critical yet complex phase in the implementation process of integrated circuits (ICs), often necessitating considerable time and effort. Given its complexity, the routing process is typically divided into two stages: 1) global routing and 2) detailed routing. Global routing, the initial stage, establishes coarse-grained wire paths for signal nets, thereby providing valuable guidance for the subsequent detailed routing stage, enhancing its efficiency. Detailed routing, on the other hand, focuses on identifying valid physical paths, primarily within the routing guides set by global routing, while taking into account design rule constraints.

In addition to guiding detailed routing, global routing also plays an important role in earlier stages of the IC implementation flow, such as logic synthesis and physical planning, where it facilitates routability and timing estimation [1]. This estimation assists in generating physical-friendly netlists during logic synthesis and aids in partitioning, I/O planning, and timing budgeting during physical planning [1]. Given the purpose of estimation, global routers for early stages have the following three characteristics [1]. First, they must be capable of handling extremely large designs (up to 100M cells) that have not yet been partitioned. Second, as a frequently used engine for routability and timing estimation, the global routers need to be highly efficient. The intensive use can result in significant runtime overhead if the routers are not fast enough. Lastly, global routers do not need to resolve congestion with high effort, because low-effort global routing results serve sufficiently for estimation purposes. These characteristics present significant scalability challenges in both design size and runtime for existing global routers.

Graphics processing unit (GPU) computing has emerged as a promising solution to the runtime scalability issue due to its support of higher parallelism and memory bandwidth. Many GPU-accelerated algorithms have demonstrated significantly improved efficiency in various Electronic design automation (EDA) problems [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. Among these works, there are three related to global routing. GAMER [10] reformulates maze routing as a prefix sum/min problem that can be efficiently solved with GPU, achieving significant speedup when integrated into the state-of-the-art global router CUGR [21]. FastGR [12] proposes a GPU-accelerated method for pattern routing and a heterogeneous task graph scheduler to improve speed. Lin and Wong [11] introduce a full-scale GPU-accelerated global router with multiple parallelization techniques, achieving more than 13× speedup over multithreaded CUGR [21].

Despite significant improvement in global routing efficiency using GPU, the scalability issue in design size remains a challenge for modern GPU-based global routers due to two reasons. First, GPU memory is limited. This requires memory-efficient solutions that can minimize CPU-GPU communication while maximizing GPU utilization. Second, large designs have more nets with bigger routing graphs, providing many new parallelization opportunities that have not yet been explored. To overcome these problems, we propose new practical techniques to parallelize large-scale global routing.

This article is an extension to the preliminary version [22]. Our contributions are summarized as follows.

1) We propose a new efficient method for batch generation. This method is based on 3-D fine-grained overlap checking and explores more parallelism by increasing

Received 22 January 2025; revised 13 May 2025; accepted 21 May 2025. Date of publication 26 May 2025; date of current version 6 January 2026. This work was supported in part by the Grant from the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project CUHK14218422. This article was recommended by Associate Editor Y. Lin. *(Corresponding author: Liang Xiao.)*

Liang Xiao, Qinkai Duan, Tsung-Yi Ho, and Evangeline F. Y. Young are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: lxiao23@cse.cuhk.edu.hk; qkduan25@cse.cuhk.edu.hk; tyho@cse.cuhk.edu.hk; fyyoung@cse.cuhk.edu.hk).

Shiju Lin is with the Hong Kong University of Science and Technology (Guangzhou), Guangzhou 511400, China (e-mail: shijulin@hkust-gz.edu.cn).

Jinwei Liu is with the Department of Computer Science, Hong Kong Baptist University, Hong Kong (e-mail: jinweiliu@comp.hkbu.edu.hk).

Digital Object Identifier 10.1109/TCAD.2025.3573685

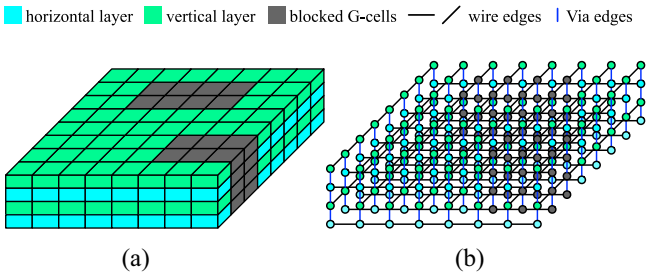


Fig. 1. Global routing grid graph. (a) G-cell partitioning. (b) Grid graph.

the number of nets per batch (nets in a batch can be routed in parallel).
2) We also propose a GPU version of the batch generation algorithm which is three times faster than the CPU version.
3) We propose a node-level parallel routing approach that achieves much higher parallelism compared to traditional net-level parallel routing.
4) We improve directed acyclic graph (DAG) routing by allowing layer changes along edges, which gives DAG routing additional flexibility to improve the quality.
5) Based on the above techniques, we have developed a GPU-based scalable global router. Our global router outperforms the winner of the GPU/ML-Enhanced Large Scale Global Routing contest in the International Symposium on Physical Design (ISPD) 2024 in both runtime and quality. The results demonstrate the effectiveness of our proposed techniques in practice.

The rest of this article is organized as follows. We first present the preliminaries in Section II. Next, we introduce two scalable techniques for parallelizing large-scale global routing in Sections III and IV. Then in Section V, we introduce a flexible layer transition (FLT) technique that enhances the original DAG routing and improves the quality. Lastly, Section VI shows the experimental results followed by the conclusions in Section VII.

II. Preliminaries

A. Global Routing Formulation

In global routing, the multilayer routing space is divided into a set of global cells (G-cells) [Fig. 1(a)]. A grid graph $G(V, E)$ can be obtained by regarding each G-cell as a vertex $v \in V$ and creating an edge $e \in E$ between every two adjacent G-cells [Fig. 1(b)]. Edges between same-layer and different-layer G-cells are called *wires* and *vias*, respectively. Note that each layer has a dedicated (horizontal or vertical) routing direction that wires should follow. A net consists of multiple pins located in different G-cells. The objective of global routing is to connect all pins of each net using wire and via edges while minimizing certain metrics, such as wirelength, via count, overflow, runtime, etc.

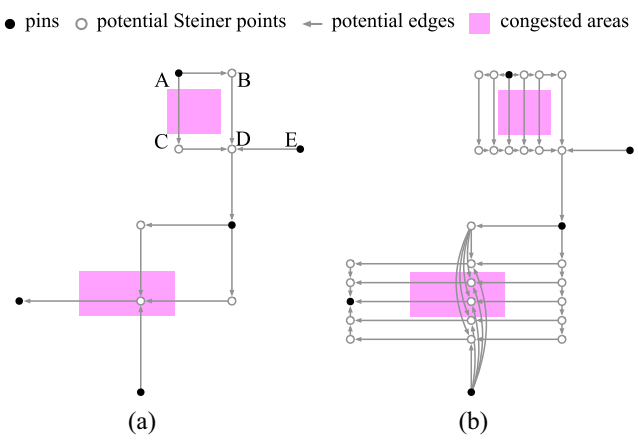


Fig. 2. Global routing using routing DAG. (a) Basic routing DAG. (b) Augmented routing DAG.

B. DAG-Based Global Routing

Our proposed parallel algorithm is mainly based on the DAG-based global routing algorithm in CUGR2 [23]. When constructing DAG, a 2D topology is first built, typically using tree construction algorithms [24], [25], [26]. In our implementation, we employ rectilinear Steiner minimum tree (RSMT) to determine the initial 2D topology for each net and build a DAG based on it. Fig. 2(a) shows a basic routing DAG for L-shape pattern routing constructed using rectilinear Steiner minimum tree. Then, we can use a dynamic programming algorithm to find the minimum cost 3-D routing topology within the DAG efficiently. If some congested areas can be detected before routing, it is also possible to augment the routing DAG by adding alternative paths for congested edge segments so as to facilitate congestion avoidance as illustrated in Fig. 2(b).

Our proposed parallel routing scheme mainly consists of two stages, initial routing, and rip-up and rerouting. In initial routing, we construct a basic routing DAG to perform L-shape pattern routing. In rip-up and rerouting, we use the initial routing results to identify congestion and augment the routing DAG accordingly before rerouting.

The advantage of adopting the DAG-based routing style for our parallel scheme is mainly two-fold. First, the routing DAG limits the search space for each net and reduces racing conditions. Second, the routing DAG can also be used to accurately identify the resources that each net will occupy when routing, which enables us to detect routing conflicts in a more fine-grained level. In our parallel scheme, we propose a novel and extremely efficient batch generation algorithm to maximize the number of nets in each batch so as to boost net-level parallelism and reduce runtime. Besides, we also propose a node-level parallel scheme that allows us to even parallelize the calculation workload of a whole routing DAG. What's more, we also propose an algorithm to enable layer changes along edges in DAG, which increases the flexibility of the original DAG routing and improves the quality.

C. Overall Flow

Our algorithm mainly consists of two parts: 1) initial routing and 2) augmented rip-up and reroute. Before routing, we break

large nets with more than 12 pins into independent small subnets to achieve higher parallelism and improve efficiency. In the first phase, we route on the basic routing DAG as shown in Fig. 2(a) to quickly generate an initial solution. In the second phase, we rip up the congested nets and reroute each net on an augmented DAG to avoid congestion. Our novel batch generation algorithm is used in both phases to increase net-level parallelism. Our node-level parallelism algorithm and the FLT algorithm are used in the rip-up and reroute phase to improve the efficiency and the quality.

D. Evaluation Metrics

In the ISPD2024 contest, routing results are evaluated by a weighted sum of the total wirelength, via count, and overflow penalty [1],

$$\frac{0.5}{M2 \text{ pitch}} \cdot \text{TotalWL} + 4 \cdot \text{ViaCount} + \text{OverflowScore} \quad (1)$$

where TotalWL and ViaCount denote the sum of the wirelength for all signal nets and the total number of vias, respectively. The overflow cost for a GCell edge with routing capacity c and routing demand d at the lth layer is calculated as follows:

$$\begin{aligned} \text{OverflowCost} \quad (c, d, l) &= \text{OFWeight[l]} \cdot e^{s(d-c)} \\ s &= \begin{cases} 0.5, \text{ if } c > 0 \\ 1.5, \text{ if } c \leq 0. \end{cases} \end{aligned} \quad (2)$$

where s is a predefined scaling factor. A large s is assigned to GCell edges with zero capacity to penalize the use of obstructed GCell edges. *OFWeight*[l] is the overflow weight for GCell edges at the lth layer, which is given. *OverflowScore* is the summation of the overflow costs for all GCell edges.

III. Net-Level Parallelism

In this section, we present our approach to achieve massive net-level parallelism by an efficient routing-graph-based overlap checking. We first review existing net-level parallelization methods in Section III-A. Next, we will describe in Section III-B our new representation for routing graphs that naturally allows an efficient graph-based exact overlap checking. Finally, we will present efficient algorithms for overlap checking based on our routing graph representation in Section III-C.

A. Net-Level Parallelism

Net-level parallelism is a common technique to parallelize global routing. It refers to simultaneous routing of a batch of nets that do not "overlap" (i.e., do not use the same routing resources). For example, nets 1 and 2 in Fig. 3 can be routed in parallel if they are routed within their own bounding boxes, because their bounding boxes do not overlap. It is extensively used in modern global and detailed routers [10], [11], [12], [21], [27], [28], [29], as most routing algorithms will restrict the routing graph for a net to enhance efficiency. For instance, Fig. 4(c) and 4(d) show two example routing graphs of the 4-pin net depicted in Fig. 4(a) [with RSMT in Fig. 4(b)], both of which occupy only a small portion of the entire routing region (51 and 98 out of 480 G-cells in Fig. 4(c) and (d), respectively).

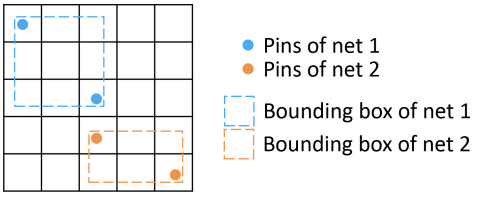


Fig. 3. Example of net-level parallelism: if nets 1 and 2 are routed within their respective bounding boxes that do not overlap, they can be routed in parallel.

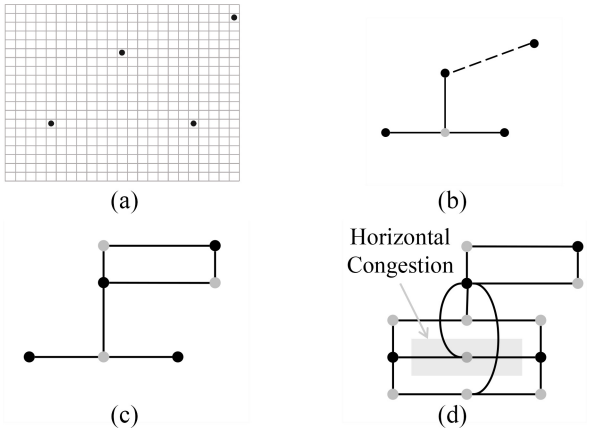


Fig. 4. Two routing graph examples. (a) Four-pin net. (b) RSMT (incomplete, dashed edge shape to be determined). (c) RSMT-based L-shape routing graph. (d) RSMT-based augmented L-shape routing graph [23].

Algorithm 1 Typical Batch Generation Algorithm

Input: a set of nets N
Output: batch count m, and batches of nets B_i $(i = 1, ..., m)$

```
1:  m ← 0
2:  for net n ∈ N do
3:      for batch_id ← 1 to m do
4:          if no overlap between n and B_batch_id then
5:              add net n to batch B_batch_id
6:              break
7:          end if
8:      end for
9:      if net n has not been added to any batch then
10:         m ← m + 1
11:         create B_m ← {n}
12:     end if
13: end for
```

Net-level parallelism is typically implemented with *batch generation* and *overlap checking*. Batch generation uses overlap checking engines to divide all the nets into batches, each of which consists of nonoverlapping nets that can be routed in parallel.

Batch Generation: Algorithm 1 shows a typical batch generation approach used by many routers [10], [11], [27], [28], [29]. For each net n, Algorithm 1 finds a batch that does not overlap with net n (lines 3–8), or creates an empty batch if such batch does not exist (lines 9–11), and insert net n into the batch.

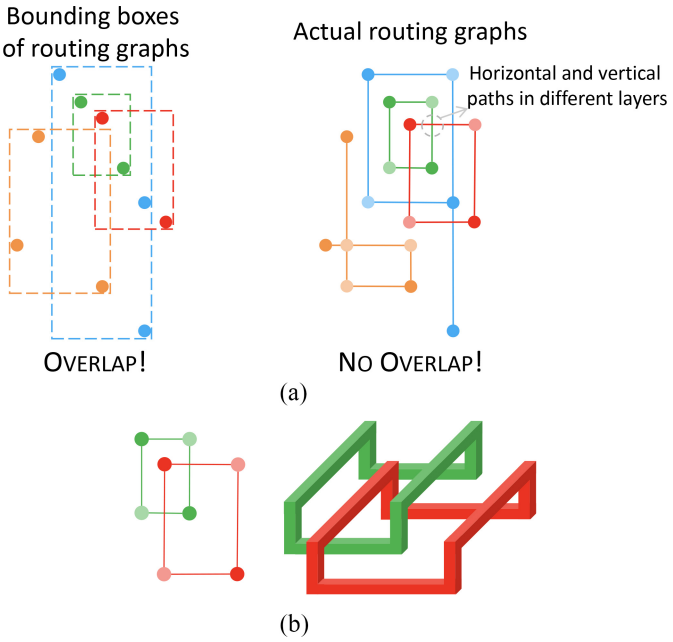


Fig. 5. Net-level parallelism for L-shape routing. (a) For RSMT-based L-shape pattern routing, the routing graphs of these four nets do not overlap but their bounding boxes overlap. (b) 3-D view to illustrate no overlap between horizontal and vertical paths (only showing two layers with horizontal and vertical routing directions for the bottom and top layers, respectively).

Overlap Checking: The bounding box is the minimum rectangle covering a routing graph, which pessimistically approximates the routing graph. This approximation is popular for overlap checking used by many routers [10], [11], [12], [21] because of the simplicity of rectangular shapes. To efficiently check the overlap of a net and a batch of nets (line 4 of Algorithm 1), R-trees are the most efficient data structure. However, the use of bounding boxes and R-trees have the following drawbacks. First, the pessimism of bounding box approximation significantly lowers the degree of parallelism. Second, complex data structures, such as R-trees are still needed even when the routing graphs are approximated by some simple bounding boxes. To address these drawbacks, we introduce a new segment-based accurate representation for routing graphs that can achieve massive net-level parallelism. Fig. 5(a) shows four nets with nonoverlapping L-shape routing graphs[1] but their bounding boxes are pairwise overlapped. These four nets will be divided into just one batch based on our exact representation of routing graphs for overlap checking, while into four batches by the traditional bounding box-based pessimistic approximation. Compared with R-trees for box overlap query, our new representation is more straightforward, allowing us to simplify the overlap checking of routing graphs into a 1-D segment overlap problem. Based on some useful observations, we will discuss our efficient overlap checking algorithm tailored for global routing scenarios in the following sections.

[1]For the "overlap" circled in dash line in Fig. 5(a), there is actually no overlap because horizontal and vertical wires are on different metal layers, as shown in Fig. 5(b).

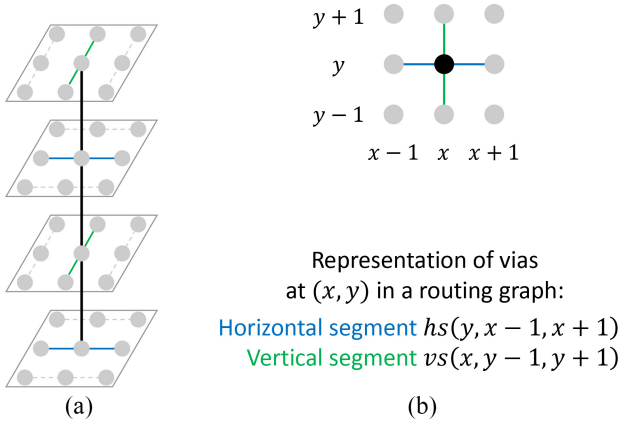


Fig. 6. Via representation by our routing graph segments. (a) Modeling vias as wire demands [1]. (b) Segment representation of vias.

B. Routing Graph Representation

In the formulation of the ISPD2024 contest [1], the routing resources are all wires, and the use of both wires and vias consumes wire resources. Our representation of routing graphs uses horizontal and vertical segments. Such a representation can be obtained by converting all possible wires and vias of the routing graph into segments according to their routing resources usage. We formally define the representation below.

Definition 1 (Segment): A horizontal segment, denoted by $hs(y, x_l, x_r)$, represents all horizontal wires between (x_l, y) and (x_r, y) in horizontal routing layers. A vertical segment $vs(x, y_l, y_r)$ is defined similarly.

Definition 2 (Routing Graph): A routing graph $G = (HS, VS)$ is represented by a set of horizontal segments HS and a set of vertical segments VS.

Definition 3 (Segment Overlap): Two horizontal segments, $hs(y_0, x_{0_l}, x_{0_r})$ and $hs(y_1, x_{1_l}, x_{1_r})$, overlap if $[x_{0_l}, x_{0_r}]$ and $[x_{1_l}, x_{1_r}]$ overlap,[2] and $y_0 = y_1$. Overlap of vertical segments is similarly defined. There is no overlap between horizontal and vertical segments, because they are on different layers.

Definition 4 (Routing Graph Overlap): Two routing graphs $G_0 = (HS_0, VS_0)$ and $G_1 = (HS_1, VS_1)$ overlap if there exist two overlapping horizontal segments $hs_0 \in HS_0$ and $hs_1 \in HS_1$ or two overlapping vertical segments $vs_0 \in VS_0$ and $vs_1 \in VS_1$.

We will illustrate how to construct such a representation for a routing graph using Fig. 4(c) as an example. For each of the wire segments in Fig. 4(c), we construct a corresponding horizontal or vertical segment. For each of the possible via locations [there are seven such locations in Fig. 4(c)], we construct horizontal segments and vertical segments to account for its wire demands as shown in Fig. 6, according to the via model described in [1]. These two simple steps complete the process of generating a representation for the L-shape routing graph in Fig. 4(c). The representation of any given routing graph can be generated in a similar fashion.

[2]Formally speaking, $[x_{0_l}, x_{0_r}]$ and $[x_{1_l}, x_{1_r}]$ overlap if there exists an integer x s.t. $x_{0_l} \le x \le x_{0_r}$ and $x_{1_l} \le x \le x_{1_r}$.

Algorithm 2 Point Exhaustion

```
1: initialize [b_1, ..., b_n] = [0, ..., 0]
2: procedure INSERT(s)                          ▷ segment s = [l, r]
3:     for i ← l to r do
4:         b_i ← 1
5:     end for
6: end procedure
7: procedure QUERY(s)                           ▷ segment s = [l, r]
8:     for i ← l to r do
9:         if b_i equal to 1 then
10:            return overlap
11:        end if
12:    end for
13:    return no overlap
14: end procedure
```

C. Efficient Overlap Checking Algorithms

In this section, we will introduce our efficient algorithms for graph-based overlap checking. For convenience, we will explain the algorithms with horizontal segments on an $X \times Y$ grid graph. Vertical segments can be handled similarly.

According to Definition 3, one necessary condition for two horizontal segments $hs(y_0, x_{0_l}, x_{0_r})$ and $hs(y_1, x_{1_l}, x_{1_r})$ to overlap is $y_0 = y_1$. Therefore, we will group the segments with the same y and the overlap checking problem becomes a 1-D overlap checking problem of $[x_{0_l}, x_{0_r}]$ and $[x_{1_l}, x_{1_r}]$. We need a data structure S supporting the following operations to facilitate overlap checking in batch generation (Algorithm 1).

1) *Insertion:* Insert a segment $s = [x_l, x_r]$ to S.
2) *Query:* Given a segment $s_q = [x_l, x_r]$, check if s_q intersects with any segment $s \in S$.

This is a classical computational geometry problem that can be efficiently solved by segment trees [30] in $O(\log n)$ time for both operations, where n is the length of the range involved. However, in the context of routing graph overlap checking, we have developed faster algorithms by leverage the following observations.

Observation 1 (Short Segments): In the largest design of the ISPD 2024 contest [1], the average horizontal segment length in the RSMTs is only 12 on a 9245×12544 grid graph. Long wires lead to large signal delay, and, hence, are optimized in stages before routing, such as placement.

Since segments are very short, we can simply use a Boolean array to record whether each point in $[1, n]$ is covered by some segment $s \in S$. We mark every point $x \in [l, r]$ when a segment $[l, r]$ is inserted, and check every point $x \in [l_q, r_q]$ for overlap query of a segment $[l_q, r_q]$. This *point exhaustion* approach is simple yet efficient because of the short average length of the segments. We show the pseudo code of *point exhaustion* in Algorithm 2.

We can further improve the efficiency of this point exhaustion by using bit arrays, which are effective at exploiting bit-level parallelism in hardware to perform operations quickly [31] (also known as bitsets). Take Algorithm 2 with $n = 64$ as an example. We can use one 64-bit integer i to store the Boolean values of $[b_1, \ldots, b_{64}]$. When a segment, say $[1, 16]$, is inserted, we can simply perform a bitwise OR operation between i and 65535, whose binary representation is 48 leading 0's followed by 16 trailing 1's. This $O(1)$ operation is equivalent to setting the least significant 16 bits of i to 1. Similarly, for a query, we can use a bitwise AND operation to check if a certain bit range of i, corresponding to the query segment, has any 1. This technique helps reduce the processing time of long segments and has demonstrated its high efficiency experimentally, which will be shown in Section VI-A.

Observation 2 (More Queries Than Insertions): As shown in Algorithm 1, each net needs multiple queries (line 4) until a nonoverlapping batch is found while performing only one insertion (line 5/10) to the batch found. The actual query-insertion ratio depends on the overlap rate affected by both benchmarks and algorithms. Preliminary experiments show $30\times$ more queries than insertions in the largest design of the ISPD2024 contest benchmarks.

Based on this observation, we develop a fast algorithm, *representative point exhaustion*, for nonexact overlap checking, which achieves increased parallelism and reduced runtime by allowing a little bit of overlap. Representative point exhaustion is identical to point exhaustion except that it only checks the two end points of a query segment. This simplified algorithm for answering queries greatly reduces the number of points that need to be checked while covering most overlap scenarios in practice. The only scenario that this algorithm fails to find the overlap of two overlapping segments is when the query segment $[l_q, r_q]$ contains the overlapping segment $[l, r]$, $[l, r] \subset [l_q, r_q]$ (but not vice versa). Empirically, there is only about 2% overlap in length by representative point exhaustion. Batch generation using representative point exhaustion offers higher parallelism in shorter runtime, which is useful for routing algorithms that are insensitive to a little overlap.

The batch generation algorithm is used in both the initial routing phase and the rip-up and reroute phase we introduced in Section II-C. The segments needed for batch generation in these two phases come from the initial DAG and the augmented DAG, respectively.

D. GPU-Accelerated Batch Generation

The batch generation algorithm (Algorithm 1) equipped with representative point exhaustion-based overlap checking (Algorithm 2) runs efficiently on CPU. However, GPU acceleration has the potential to bring the efficiency to another level. In this section, we introduce our newly proposed GPU-accelerated batch generation algorithm.

Accelerating the CPU version of the batch generation algorithm has the following challenges.

1) *Net Conflicts:* When distributing nets across multiple threads for concurrent batch assignment, conflicting nets may be erroneously assigned to the same batch, compromising the fundamental nonoverlap requirement.
2) *Batch Efficiency:* Another approach is to partition nets into separate groups and use different threads to generate independent batches for each group. However, that means nets from different groups can never be assigned to the same batch. This may lead to insufficient utilization of each batch's resources. The total number of

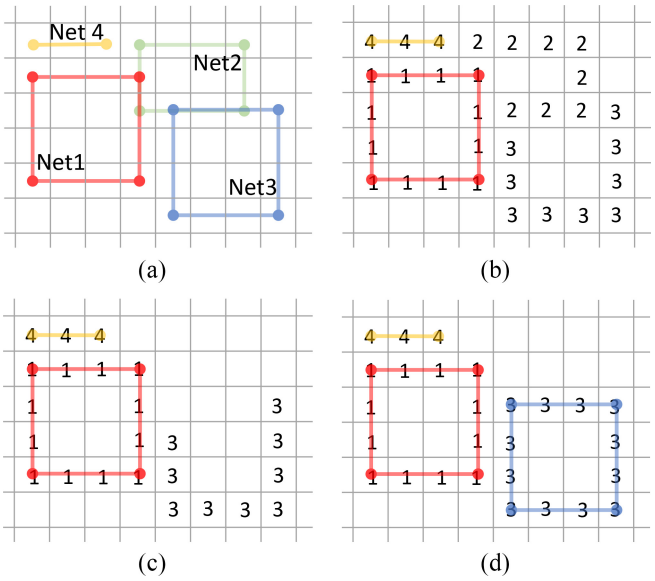


Fig. 7. Example of GPU-accelerated batch generation. (a) The input four nets of batch generation. (b) The first commit phase. Nets 1 and 4 commit all segments to the map successfully, so they are assigned to this batch. Some segments of Nets 2 and 3 are occupied by nets with higher priority, so they cannot be assigned to this batch currently. (c) Net 2 has conflicts with an assigned net (Net 1), so it is ruled out of this batch. (d) In the second iteration, Net 3 successfully commits all segments and has no conflict with other nets. Thus, it is assigned to this batch.

batches may increase and the overall routing time may extend.

3) *Net Order:* Net ordering significantly impacts routing quality. Critical nets and less flexible nets (e.g., shorter nets) should be prioritized in routing resource allocation to optimize wirelength and reduce congestion. While CPU-based sequential routing naturally processes nets in priority order, maintaining such precedence becomes challenging in parallel batch generation.

Given the above challenges, it is evident that simply partitioning nets for different threads for parallel processing is insufficient. To address these challenges, we propose a priority-based conflict resolution mechanism where nets with higher priority can override the ones with lower priority.

More specifically, we will create one new batch at a time and try to commit the segments of all nets to the batch in parallel. When two segments from different nets overlap, the one with the higher priority will override the other one. After all segments are committed, we will check each net in parallel. If all the segments of a net are successfully committed, the net will be assigned to the current batch. If some segments of a net conflict with the assigned nets, the net will be ruled out for consideration in the current batch and their segments will be un-committed. We will repeat the commit-check cycle until all nets are either assigned or ruled out. Then, we will create a new batch to accommodate the remaining nets. The procedure continues until all nets are assigned to batches.

Fig. 7 illustrates this process with four nets, where lower index numbers indicate higher priority. First, all three nets will try to commit all segments to the map of this batch as shown in Fig. 7(b). Since Net 1 and 4 successfully committed all segments on the map, they are assigned to this batch. However, some segments of Net 2 are occupied by Net 1 and some segments of Net 3 are occupied by Net 2, so they cannot be assigned to this batch currently. Second, we will check nets to find those who have conflicts with an assigned net. Such nets will have no chance to be assigned to this batch. So we will rule out these nets from this batch to give space to nets with lower priority. For example, the segments of Net 2 is cleared out due to conflicting with Net 1 in Fig. 7(c). Then we complete a commit-check iteration. As shown in Fig. 7(d), in the next iteration, Net 3 can commit all segments to this batch. Thus it is assigned to this batch. We repeat these commit-check operations until no more nets can be assigned to this batch. Then, we will accommodate the remaining net in later batches.

While our GPU implementation produces similar results to the CPU version, it achieves approximately a threefold speedup. The batch number and runtime are further discussed in Section VI-A.

IV. Node-Level Parallelism

A. Node-Level Parallelism

To accelerate the dynamic programming on an augmented DAG, a straightforward approach might be to route all the nodes simultaneously. However, in the DAG-based routing, a node's cost is calculated by aggregating its incoming nodes' costs [For example, the cost of node a is partially determined by the costs of nodes b and c in Fig. 8(b)]. There is thus a dependency relationship between nodes. We model this dependency as "depth." The root node will have a depth of 0, and the depth of any node is one deeper than its deepest outgoing nodes. By routing nodes of the same depth in parallel, we can achieve node-level parallelism.

Fig. 8 illustrates an example of how we route nets batch by batch with node-level parallelism. Suppose we have 4 nets, Net A, B, C, and D in our grid graph. Since nets with overlap cannot be routed together, Net A and B are distributed to batch 0, as shown in Fig. 8(a), and nets C and D are distributed to batch 1. Fig. 8(c) shows the task distribution of a net-level parallel strategy, i.e. routing each net by one thread. Let $D_0 \sim D_6$ denote depth 0 to 6, $T_0 \sim T_1$ denote thread 0 to 1. We can only utilize 2 threads to route 2 nets, and a total of 7 steps are needed to finish batch 0, and a total of 4 steps are needed to finish batch 1. However, in Fig. 8(c), by routing nodes with the same depth simultaneously, we only need 4 steps to finish batch 0 and 3 steps to finish batch 1. In total, we need 7 steps to route with node-level parallelism, which is 4 steps fewer than the net-level parallel strategy.

To calculate the depths of the nodes, we can make use of a recursive algorithm similar to the one finding a topological sort in a graph. First, we initialize the depth of the root node to 0. Then, we compute the depth of the remaining nodes recursively. When the depths of all the outgoing nodes of a node are determined, we set the depth of the current node as one plus the maximum depth of all its outgoing nodes.

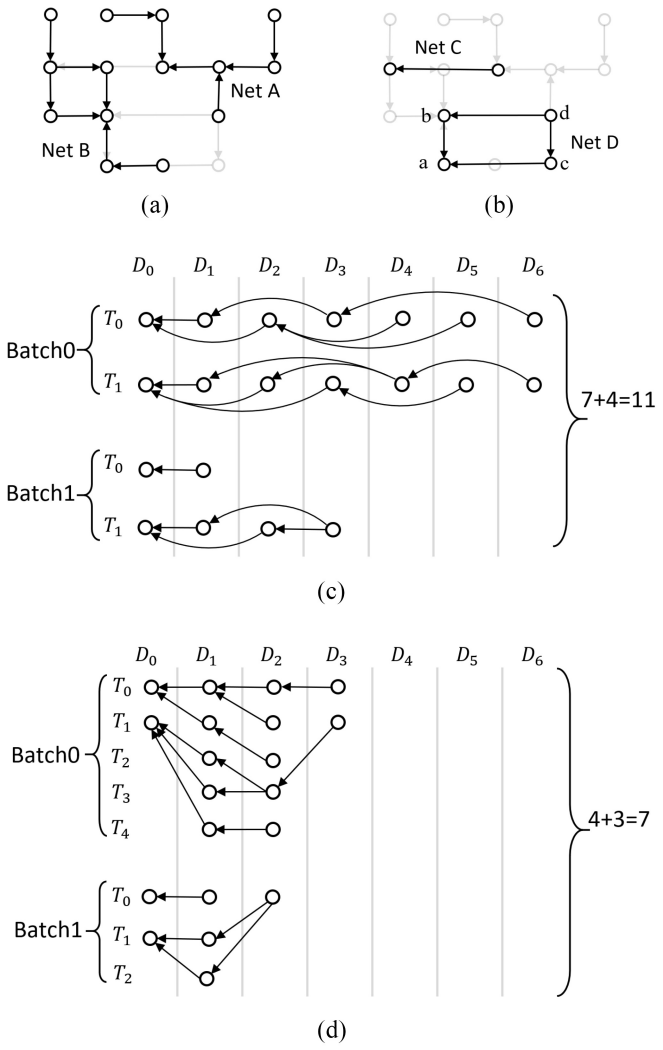


Fig. 8. Example of node-level parallelism. (a) Batch0. (b) Batch1. (c) w/o node-level parallelism. (d) w/ node-level parallelism.

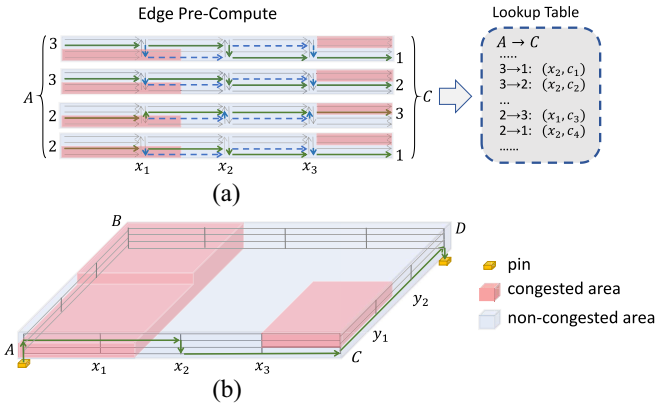


Fig. 9. Example of 3-D routing with FLT. (a) Lookup table generation. Blue dashed lines are candidate connections of a given starting and arrival layer pair. Green lines are the connections with the lowest cost (denoted by $c_1, c_2, \ldots$, in the figure). Best costs and connections are computed in parallel and then stored in a lookup table. For example, $(3 \rightarrow 1 : (x_2, c_1))$ denotes that the best scheme of connecting (node A, layer 3) to (node B, layer 1) is changing layer at x_2 with cost c_1. (b) A sample solution for the 2-pin net. It changes layer once to avoid the congested areas.

V. Flexible Layer Transition Algorithm

In this section, we propose an algorithm that enables layer changes along edges in DAG routing. It gives DAG routing additional flexibility to improve the quality. Hereafter, this approach is referred to as the FLT algorithm.

As mentioned above, our acceleration techniques are largely based on CUGR2 [23]. Although this DAG routing algorithm achieves good performance and handles most cases effectively, we observed that it could be further enhanced to handle extremely congested situations. One limitation of the original DAG routing is that two nodes in the routing DAG must be connected by a single wire segment on a certain layer. No layer change is allowed along the wire segment.

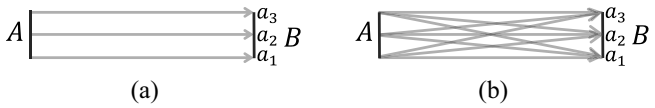
Fig. 9(b) gives an example of a very congested scenario. The routing graph consists of four horizontal layers and three vertical layers, while the 2-pin net has two alternative directions to connect the pins ($A-C-D$ and $A-B-D$ are two optional paths). So in total, we have 14 wire segments to use. However, none of the 14 segments is congestion-free, which means if we do not allow layer changes along a single segment, overflow is unavoidable. Nevertheless, by allowing one layer change at x_2 on edge $A \rightarrow C$, the final path can avoid all congested areas.

In the original DAG-based routing, the routing cost of a node N on a specific layer l, $cost(N, l)$, is computed recursively as a sum of,

1) The routing cost of each incoming node M in the DAG on certain layer l_m, $cost(M, l_m)$.
2) The wire cost to connect each incoming node to the current node on layer l_m.
3) The via cost needed to connect all wires to the desired end layer l at node N.

Since no layer change is allowed along the wire edges in the original DAG-based routing, we only need to choose a layer index l_m for each incoming node M to minimize the overall routing cost. In order to allow layer changes along the wire edges, our FLT algorithm also considers situations where edges may start and end on different layers. For example, the connection between A and C in Fig. 9(b).

We illustrate the difference between the connection approach of the original DAG routing and our new approach in Fig. 10. Suppose we have two nodes, node A is an incoming node of node B. The minimum costs of reaching B from A on layer 1 to 3 are a_1, a_2, a_3 respectively. In the original implementation, we only try connecting using a wire segment on one layer as shown in Fig. 10(a). For example, $a_1 = cost(A, 1) + wireCost(A \rightarrow B, 1)$, where $wireCost(\cdots)$ is the cost of the wire segment on layer 1 connecting A and B. To improve the flexibility, now we allow connections that start and end on different layers as shown in Fig. 10(b). For example, $a_1 = min_{i=1,2,3}(cost(A, i) + edgeCost(A \rightarrow B, i \rightarrow 1))$, where $edgeCost(\cdots)$ is the precomputed cost of connecting A and B using wire that starts on layer i and ends on layer 1. Note that a_1, a_2, a_3 are only partial costs, and the final routing cost of B should also consider connections from other incoming nodes and via costs.



(a) (b)

Fig. 10. Comparison of the original and the new way of connecting two nodes. (a) The original way of connecting node A to node B. (b) The new way of connecting the two nodes. The minimum costs of reaching B from A on layer 1 to 3 are a_1, a_2 and a_3, respectively.

Algorithm 3 Edge Precompute

```
1: procedure PRECOMPUTE(u, v, l_start, l_arr)
2:     p_0, p_1, ..., p_n ← Interpolate(u, v, d)
3:     for p ← p_0, p_1, ..., p_n do
4:         cost1 ← wire(u → p, l_start)
5:         via_cost ← via(p, l_start → l_arr)
6:         cost2 ← wire(p → v, l_arr)
7:         cost ← cost1 + cost2 + via_cost
8:         if cost < edgeCost(u → v, l_start → l_arr) then
9:             edgeCost(u → v, l_start → l_arr) ← cost
10:            changePoint(u → v, l_start → l_arr) ← p
11:        end if
12:    end for
13: end procedure
```

Obviously, in the original DAG routing, a node needs to enumerate different layers for each incoming node, which takes $O(L)$ time, where L is the number of layers. The new strategy of enumerating different combinations of starting and arrival layers will increase the time needed to $O(L^2)$. However, this won't worsen the overall time complexity, as finding the best combinations of layers to connect different incoming nodes also takes at least $O(L^2)$ time, using the method in [23].

To efficiently compute the best connection scheme of two nodes given the starting and arrival layers, we introduce a preprocessing step in Algorithm 3 (namely, Edge Precompute). In our implementation, we only allow each edge to change layer at most once to keep things simple and limit the number of vias. Thus, to find the best connection scheme, we only need to find the best layer change point. The computation is performed using GPU parallelization, where each thread computes the optimal connection scheme for one pair of starting and arrival layers for one edge. After this precomputation, we will know the best layer change point and the cost of connecting the two layers of the two nodes.

We use Algorithm 3 to compute the best connection schemes for all edges, starting and arrival layers in parallel. For each edge, We first interpolate the candidate layer changing points $P = (p_0, p_1, \ldots, p_n)$ between the two nodes.

In our implementation, equidistant interpolation at distance d is performed. For an edge traversing k G-Cells, d is set to $\max\{\lfloor(k/20)\rfloor, 1\}$. For an edge with a length less than 20, each point on the edge is a candidate layer change point. Otherwise, we choose from the $\lfloor(k/d)\rfloor$ equidistant points.

Since we allow only one layer change per edge, the total cost is the sum of the cost of the two segments partitioned by point p (i.e. the wire cost from u to p on the starting layer l_{start}, and p to v on the arrival layer l_{arr}) and the via cost for changing layers at node p. We check all possible points p and record the best scheme.

Each GPU thread will process all points along an edge. Therefore, one batch needs $O(len(e_{\max}))$ time to finish, where $len(e_{\max})$ is the length of the longest edge in a batch. This preprocessing helps us quickly build a lookup table that stores the minimum wire costs of connecting two nodes with different starting and arrival layers. Since all edges are computed in parallel, edge precompute is very fast and does not take too much time, which we will demonstrate in Section VI-D.

To sum up, our FLT algorithm extends the original DAG-based routing by considering more flexible connections between nodes. Though inserting additional nodes along the edges in DAG could also enable layer changes along edges, that approach would significantly increase the routing tree depth and potentially increase runtime dramatically. In contrast, our proposed FLT algorithm efficiently explores many layer change points through parallel computation, maintaining computational efficiency while providing greater routing flexibility.

TABLE I
BENCHMARK DETAILS [1].

	Benchmark (BM)	#Nets	#Pins	Gcell Grid
0	Ariane_sample	129K	420K	844×1144
1	MemPool-Tile_sample	136K	500K	475×644
2	NVDLA_sample	177K	630K	1240×1682
3	BlackParrot_sample	770K	2.9M	1532×2077
4	MemPool-Group_sample	3.3M	10.9M	1782×2417
5	MemPool-Cluster_sample	10.6M	40.2M	3511×4764
6	TeraPool-Cluster_sample	59.3M	213M	7891×10708
7	Ariane_rank	128K	435K	716×971
8	MemPool-Tile_rank	136K	483K	429×581
9	NVDLA_rank	174K	610K	908×1682
10	BlackParrot_rank	825K	2.9M	1532×2077
11	MemPool-Group_rank	3.2M	10.9M	1782×2417
12	MemPool-Cluster_rank	10.6M	40.2M	4113×5580
13	TeraPool-Cluster_rank	59.3M	213M	9245×12544

VI. EXPERIMENTAL RESULTS

We conducted our experiments on a 64-bit Linux workstation with Intel Xeon Gold 6326 CPUs (2.90 GHz) and 256-GB memory. To be consistent with the ISPD2024 contest environment [1], all the experiments were run with 1 NVIDIA A800 GPU and 8 CPU threads. We used the same benchmarks (details shown in Table I) and evaluator as in the contest [1].

A. Overlap Checking

To demonstrate the effectiveness of our new routing graph representation and algorithms, we compare different overlap checking approaches on the two most challenging benchmarks, 12 and 13. These two benchmarks have the most number of nets and the largest grid graphs, and their experimental results are shown in Table II. We can measure the degree of parallelism by the number of batches. Since the nets in a batch can be routed in parallel, fewer batches imply more parallelism. We can see that using routing graphs to form batches are

TABLE II
BATCH GENERATION RESULTS WITH DIFFERENT OVERLAP CHECKING METHODS ON TWO LARGEST BENCHMARKS

Geometry	Method	Benchmark 12		Benchmark 13		Overlap
		#Batches	Runtime (s)	#Batches	Runtime (s)	
Bounding Box of Routing Graph	R-Tree	4748	1742	26038	36033	No
	R-Tree (1000†)	10109	281	46892	5322	No
Segment-Based Routing Graph (Ours)	Segment Tree	515	199	554	1662	No
	Point Exhaustion	515	43	554	330	No
	Point Exhaustion (Bitset)	515	40	554	307	No
	Point Semi-Exhaustion	383	20	527	121	2% Length
	Point Semi-Exhaustion(GPU)	370	6.3	515	40	2% Length

†The number of nets per batch is limited to 1000.

TABLE III
OVERALL EXPERIMENTAL RESULTS OF TOP-2 GLOBAL ROUTERS OF ISPD2024 CONTEST AND INSTANTGR

Benchmark	HeLEM-GR		1st Place Team		2nd Place Team		InstantGR 1.0		InstantGR 2.0	
	Score	Time (s)	Score	Time (s)	Score	Time (s)	Score	Time (s)	Score	Time (s)
0	19734912	2.22	19789143	3.44	20099496	2.10	19716744	2.55	**19655065**	**1.53**
1	15195055	2.1	15241650	2.95	15432389	1.97	15126510	2.56	**15094708**	**1.36**
2	48108770	3.34	48257027	5.60	48837685	3.31	47982224	3.73	**47938219**	**2.75**
3	112985593	8.55	113562198	8.20	113321049	18.49	112474880	14.64	**112149983**	**6.23**
4	396514188	**16.86**	398169317	56.59	411758419	35.53	397658013	35.78	**395530319**	23.79
5	1616064157	**79.94**	1626227314	231.87	1665748885	142.16	1623946297	116.63	**1615066379**	86.86
6	**18468687634**	1117.06	19609525592	2821.53	19684092627	2115.14	19139291553	1289.02	18493908472	**1049.77**
7	22561641	2.25	22602876	3.75	23093501	2.81	22545800	2.98	**22481586**	**1.78**
8	13773881	2.19	13827142	2.49	14133269	2.45	13774789	2.87	**13736300**	**1.72**
9	43031318	3.05	43195979	4.97	44141552	4.29	43047147	3.83	**42936806**	**2.68**
10	110140593	7.73	113109620	16.71	110986781	15.87	109844055	9.45	**109588812**	**5.59**
11	381972892	**16.87**	383637652	43.75	395488859	40.20	382639889	36.35	**380869353**	22.28
12	1775052604	**73.05**	1782191834	214.00	1824527054	148.90	1780897390	117.75	**1771528815**	80.05
13	11995425076	**644.18**	12528609838	1654.27	12676067016	1338.30	12257767067	882.66	**11970080776**	679.67
Avg. Ratio	1.003	1.110	1.016	2.160	1.031	1.742	1.007	1.563	**1.000**	**1.000**

orders of magnitude better than using bounding boxes. For example, on benchmark 13, graph-based geometry produces hundreds of batches while box-based geometry generates tens of thousands of batches. This is not surprising because routing graphs are much more accurate than bounding box approximation at the cost of having more complex geometry. However, we overcome the complexity of using graph-based geometry by our segment-based representation and efficient point exhaustion algorithms. The effectiveness of our method is confirmed by the short runtime shown in Table II, which is significantly smaller than that of using R-trees for bounding boxes or using segment trees for our representation. Our GPU-accelerated representative point exhaustion achieves 3× speedup over the CPU version.

We present the batch generation results of benchmarks 7–13 in Fig. 11 to demonstrate the scalability of our approach. When the number of nets increases, our method increases more slowly than the traditional box-based R-tree method in terms of both batch count and runtime. The exceptional scalability shows the practical usefulness of our method even when modern ICs keep increasing in size and complexity. Moreover, the GPU-accelerated point exhaustion method further improves the scalability and speeds up the CPU version by around 3× for larger benchmarks (10–13).

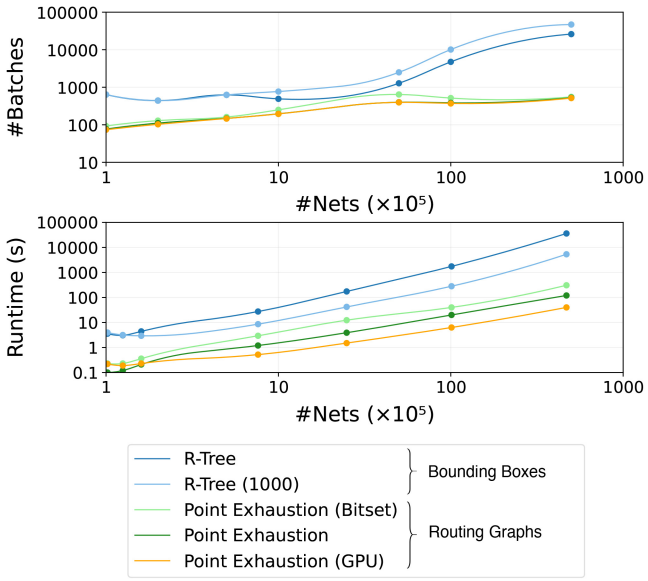


Fig. 11. Scalability comparisons of different overlap checking methods (experiments on benchmarks 7–13).

B. Global Routing

We conduct our experiments on the ISPD2024 contest benchmark suites [1] and evaluate the results using the contest's official evaluator. For a fair comparison, we obtained the binary files from the top 2 teams and the author of HeLEM-GR [32]. We evaluated the binaries on our machine using their default settings.

Table III presents the overall comparison between the contest's top 2 teams, HeLEM-GR [32], our ICCAD version (InstantGR 1.0), and the proposed version (InstantGR 2.0). In the table, the "Score" is given by the official evaluator from

TABLE IV
DETAILED COMPARISON OF THE FIRST PLACE AND THE TWO VERSIONS OF OUR ROUTER

Benchmark	WL			Via			Overflow		
	HeLEM-GR	InstantGR 1.0	InstantGR 2.0	HeLEM-GR	InstantGR 1.0	InstantGR 2.0	HeLEM-GR	InstantGR 1.0	InstantGR 2.0
0	**9333438**	9361015	9337717	2844740	2794988	**2757792**	**7556734**	7560742	7559556
1	8338598	8350182	**8332314**	3341728	3259740	**3241368**	**3514729**	3516588	3521025
2	**21242061**	21289207	21248818	4464608	4302208	**4292424**	22402101	22390810	**22396977**
3	58060585	58157063	**57957368**	19472196	18932720	**18901508**	35452812	35385097	**35291106**
4	**259468578**	260156132	259612541	72532976	71296860	**70507184**	**64512634**	66205020	65410593
5	1089016864	1091435102	**1088594805**	266228272	**256292732**	256736048	**260819021**	276218463	269735527
6	12210612912	12246523242	**12187850820**	1642840776	**1495036520**	1535268064	**4615233946**	5397731791	4770789588
7	**11937576**	11972097	11952342	2875672	2827892	**2794208**	7748393	7745811	**7735036**
8	7544046	7553701	**7538808**	3287124	3212488	**3188904**	**2942711**	3008600	3008588
9	**21537535**	21582358	21546447	4581360	4411648	**4406128**	**16912423**	17053141	16984232
10	55596948	55660842	**55534330**	19513832	18993980	**18896916**	**35029813**	35189233	35157566
11	**247576859**	248207884	247634782	71919096	70609280	**69727536**	**62476937**	63822725	63507035
12	1186752708	1189011781	**1185981071**	277553000	**267495852**	268319212	**310746896**	324389756	317228532
13	7984351151	8009203448	**7977154626**	1529353596	**1433232844**	1453985984	**2481720329**	2815330776	2538940166
Avg. Ratio	1.000	1.003	**1.000**	1.037	1.002	**1.000**	**0.988**	1.021	1.000

TABLE V
RUNTIME (S) OF DAG-BASED AUGMENTED ROUTING WITH AND WITHOUT NODE-LEVEL PARALLELISM

Benchmark		0	1	2	3	4	5	6	7	8	9	10	11	12	13	Average
Nodes†		46125	65579	52617	120041	290439	874310	13166378	63214	97591	61631	45332	298021	845910	5742828	1555001
Depth‡		4222	3988	3989	20323	18930	61654	665435	7235	5652	4794	3618	14384	59531	316005	84982
Nodes/Depth		10.92	16.44	13.19	5.91	15.34	14.18	19.79	8.74	17.27	12.86	12.53	20.72	14.21	18.17	14.30
RT(s)	net-level	5.99	7.03	7.87	13.82	40.98	136.27	2329.77	7.27	8.18	9.65	6.02	36.51	130.53	860.45	257.17
	node-level	0.65	0.65	0.58	2.14	4.25	11.58	160.53	0.91	0.86	0.72	0.64	3.42	11.17	75.53	19.54
	Acceleration	9.27	10.88	13.52	6.46	9.65	11.77	14.51	8.03	9.56	13.48	9.36	10.66	11.69	11.39	10.73

†The total number of nodes of the *bottleneck net* of all batches, where *bottleneck net* is the net with the largest number of nodes in a batch.
‡The sum of the depth of the *bottleneck net* of all batches, where *bottleneck net* is the net with the largest depth in a batch.

the contest [1], which is computed using (1). With the help of FLT algorithm, we improved the quality of our ICCAD version by 0.7%. Since our FLT algorithm can help reduce overflow, fewer nets are ripped up. Along with the time reduction brought by our GPU-accelerated batch generation algorithm, we can further achieve 1.56× acceleration. Compared to other routers, our router achieves 2.16× speedup and a 1.6% quality improvement over the contest winner, while demonstrating a 0.3% quality improvement and an 11% runtime reduction compared to HeLEM-GR.

Table IV presents a detailed score comparison between our two router versions and HeLEM-GR. The new version achieves improvements in wirelength, via count, and overflow compared to our ICCAD version. Compared to HeLEM-GR, we generate 1.2% more overflow but 3.7% less vias. That is because HeLEM-GR uses a 2-D routing-layer assignment flow. Although this flow is flexible in handling overflow, it leads to more frequent layer changes and thus more vias, which increases the difficulty for detailed routing.

C. Node-Level Parallelism

In this section, we will discuss our ablation study on node-level parallelism. For comparison, we implemented a net-level and a node-level parallel kernel function. This kernel function is invoked during the dynamic programming process of updating the costs of the nodes and tracing paths. We will invoke the net-level function once for each batch, while the node-level function will be invoked as many times as the maximum depth in a batch. Other parts of the code are the same in the two versions. The detailed results are shown in Table V.

When routing nets in batches, the net-level parallel algorithm will use a thread to route one net. Therefore, the bottleneck of each batch is the net with the highest number of nodes. This is shown in Table V, where we list the sum of node counts for the bottleneck nets in all batches (This comparison is conducted under the setting of disabling the FLT algorithm) In contrast, in our node-level parallel approach, the bottleneck for each batch comes from the net with the largest depth, which is 14.3× less than the number of nodes. The reason for the substantial ratio of nodes to depths in Table V is that during the augmented DAG routing phase, a large number of alternative paths are added to find a path that minimizes congestion as much as possible. This results in having many nodes of the same depth, which makes our node-level parallelism much more effective.

We also compare the running time of the two versions of CUDA kernel function in Table V. Experiments show that our node-level parallelism can accelerate routing efficiently, which is on average 10.7× faster than the typical net-level parallel strategy.

D. Runtime Breakdown

In this section, we will show the runtime breakdown of our program. We show the runtime breakdown of the largest case (i.e. case13) in Fig. 12. From the figure, we can see that the edge precompute we introduced in Section V accounts for 22% of the augmented routing time and 7.1% of the total runtime, demonstrating the efficiency of our edge precompute algorithm.

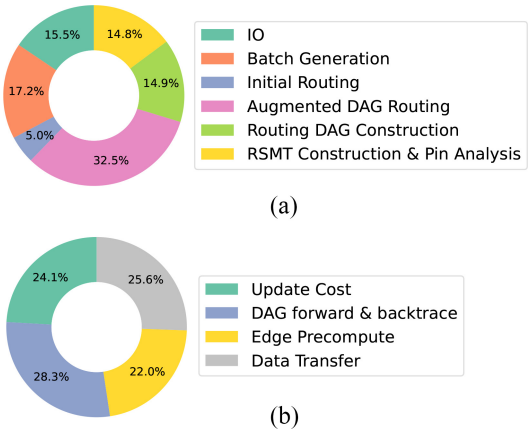


Fig. 12. Runtime breakdown. (a) Runtime breakdown of the whole program. (b) Runtime breakdown of the augmented DAG routing phase. "Update Cost" is the time spent on updating the wire usage, cost of wire, and overflow on the routing grid. "DAG forward and backtrace" is the time of updating node cost from child to parent nodes and finding paths. "Data transfer" denotes the time of transferring routing DAG in CPU to arrays in GPU.

VII. Conclusion

In this work, we propose techniques to accelerate global routing and improve its quality. We have achieved a significant improvement over the traditional bounding box-based overlap checking methods, developing a precise and efficient batch generation algorithm. It significantly increases the number of nets per batch for parallel routing, thereby fully unlocking the advantage of GPUs over CPUs. Based on our ICCAD version, we further propose a GPU-accelerated batch generation algorithm. These batch generation algorithms have the potential to be applied in many other routers that involve net-level parallelism.

To improve the original DAG-based algorithms, we also propose a node-level parallel algorithm to process nodes with the same depth, which further accelerates our program. Furthermore, we also propose a FLT algorithm, which gives DAG-based routing more flexibility to avoid congestion.

With the FLT algorithm and the GPU-accelerated batch generation, our router achieves a 0.7% quality improvement and 1.56× acceleration over our previous version. Compared to the existing state-of-the-art algorithm HeLEM-GR [32], we can also achieve a 0.3% quality improvement and a 3.7% vias reduction.

Acknowledgment

The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust.

References

[1] R. Liang, A. Agnesina, W.-H. Liu, and H. Ren, "GPU/ML-enhanced large scale global routing contest," in *Proc. Int. Symp. Phys. Design*, New York, NY, USA, 2024, pp. 269–274.

[2] S. Lin, J. Liu, T. Liu, M. D. F. Wong, and E. F. Y. Young, "NovelRewrite: Node-level parallel AIG rewriting," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, New York, NY, USA, 2022, pp. 427–432. [Online]. Available: https://doi.org/10.1145/3489517.3530462

[3] T. Liu and E. F. Young, "Rethinking AIG resynthesis in parallel," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[4] T. Liu, L. Chen, X. Li, M. Yuan, and E. F. Y. Young, "FineMap: A fine-grained GPU-parallel LUT mapping engine," in *Proc. 29th Asia South Pac. Design Autom. Conf.*, 2024, pp. 392–397.

[5] T. Liu, Y. Sun, L. Chen, X. Li, M. Yuan, and E. F. Young, "A unified parallel framework for LUT mapping vol. 44, no. 1, pp. 214–226, Jan. 2025.

[6] Y. Sun, T. Liu, M. D. Wong, and E. F. Young, "Massively parallel AIG resubstitution," in *Proc. 61st ACM/IEEE Design Autom. Conf. (DAC)*, 2024, pp. 1–6.

[7] L. Liu, B. Fu, M. D. F. Wong, and E. F. Y. Young, "Xplace: An extremely fast and extensible global placement framework," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, New York, NY, USA, 2022, pp. 1309–1314. [Online]. Available: https://doi.org/10.1145/3489517.3530485

[8] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCDPlace: Accelerated batch-based concurrent detailed placement on multithreaded CPUs and GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5083–5096, Dec. 2020.

[9] L. Liu, B. Fu, S. Lin, J. Liu, E. F. Y. Young, and M. D. F. Wong, "Xplace: An extremely fast and extensible placement framework," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 6, pp. 1872–1885, Jun. 2024.

[10] S. Lin, J. Liu, E. F. Y. Young, and M. D. F. Wong, "GAMER: GPU-accelerated maze routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 2, pp. 583–593, Feb. 2023.

[11] S. Lin and M. D. F. Wong, "Superfast full-scale GPU-accelerated global routing," in *Proc. 41st IEEE/ACM Int. Conf. Comput.-Aided Design*, 2022, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3508352.3549474

[12] S. Liu et al., "FastGR: Global routing on CPU–GPU with heterogeneous task graph scheduler," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 7, pp. 2317–2330, Jul. 2023.

[13] Z. He, Y. Ma, and B. Yu, "X-Check: GPU-accelerated design rule checking via parallel Sweepline algorithms," in *Proc. 41st IEEE/ACM Int. Conf. Comput.-Aided Design*, New York, NY, USA, 2022, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3508352.3549383

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

[15] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated critical path generation with path constraints," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, 2021, pp. 1–9.

[16] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated path-based timing analysis," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 721–726.

[17] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *Proc. 39th Int. Conf. Comput.-Aided Design*, New York, NY, USA, 2020, pp. 1–9.

[18] D.-L. Lin, Y. Zhang, H. Ren, B. Khailany, S.-H. Wang, and T.-W. Huang, "GenFuzz: GPU-accelerated hardware fuzzing using genetic algorithm with multiple inputs," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[19] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. F. Wong, "A GPU-accelerated framework for path-based timing analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 11, pp. 4219–4232, Nov. 2023.

[20] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. F. Young, and M. D. Wong, "GCS-timer: GPU-accelerated current source model based static timing analysis," in *Proc. 61st ACM/IEEE Design Autom. Conf.*, New York, NY, USA, 2024, pp. 1–6.

[21] J. Liu, C.-W. Pui, F. Wang, and E. F. Y. Young, "CUGR: Detailed-routability-driven 3D global routing with probabilistic resource model," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[22] S. Lin, L. Xiao, J. Liu, and E. F. Y. Young, "InstantGR: Scalable GPU parallelization for global routing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2024, pp. 1–8.

[23] J. Liu and E. F. Young, "EDGE: Efficient DAG-based global routing engine," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[24] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, Jan. 2008, doi: 10.1109/TCAD.2007.907068.

[25] W. K. Chow, L. Li, E. F. Y. Young, and C. W. Sham, "Obstacle-avoiding rectilinear Steiner tree construction in sequential and parallel approach," *Integr. VLSI J.*, vol. 7, no. 1, pp. 105–114, 2014, doi: 10.1016/j.vlsi.2013.08.001.
[26] L. Li, Z. Qian, and E. F. Y. Young, "Generation of optimal obstacle-avoiding rectilinear Steiner minimum tree," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2009, pp. 21–25.
[27] G. Chen, C.-W. Pui, H. Li, and E. F. Y. Young, "Dr. CU: Detailed routing by sparse grid graph and minimum-area-captured path search," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 9, pp. 1902–1915, Sep. 2020.
[28] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Y. Young, "Dr. CU 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2019, pp. 1–7.
[29] G. Chen, C.-W. Pui, H. Li, J. Chen, B. Jiang, and E. F. Y. Young, "Detailed routing by sparse grid graph and minimum-area-captured path search," in *Proc. 24th Asia South Pac. Design Autom. Conf.*, New York, NY, USA, 2019, pp. 754–760.
[30] M. Berg, O. Cheong, M. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008, doi: 10.1007/978-3-540-77974-2.
[31] Wikipedia. "Bit array—Wikipedia, the free encyclopedia." 2024. Accessed: Apr. 29, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Bit_array
[32] C. Zhao, Z. Guo, R. Wang, Z. Wen, Y. Liang, and Y. Lin, "HeLEM-GR: Heterogeneous global routing with Linearized exponential multiplier method," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2024, pp. 1–9.

Liang Xiao received the B.Eng. degree from Xi'an Jiaotong University, Xi'an, China, in 2023. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, under the supervision of Prof. Evangeline F. Y. Young.

His research interests include parallel and efficient algorithms for global routing.

Shiju Lin received the B.Eng. degree from the South China University of Technology, Guangzhou, China, in 2020, and the Ph.D. degree from the Chinese University of Hong Kong, Hong Kong, in 2024.

He is an Assistant Professor in the Microelectronics Thrust with the Hong Kong University of Science and Technology (Guangzhou), Guangzhou. His research focuses on combinatorial optimization and GPU acceleration for electronic design automation.

Dr. Lin won first place in the CAD Contest at ICCAD in 2021 and second place in the ISPD Contest in both 2021 and 2024.

Jinwei Liu received the B.Sc. degree in computer science and technology from Sichuan University, Sichuan, China, in 2018, and the Ph.D. degree from the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Hong Kong, in 2022.

He is currently an Assistant Professor with the Department of Computer Science, Hong Kong Baptist University, Hong Kong. His research interests include electronic design automation, combinatorial optimization and machine learning.

Qinkai Duan is currently pursuing the B.Eng. degree with The Hong Kong University of Science and Technology, Hong Kong. He will pursue the Ph.D. degree in August 2025.

His research focuses on timing analysis, routing algorithms, and AI applications in electronic design automation.

Tsung-Yi Ho (Fellow, IEEE) received the Ph.D. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2005.

He is a Professor with the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Hong Kong. His research interests include several areas of computing and emerging technologies, especially in design automation of microfluidic biochips.

Prof. Ho has been the recipient of the Invitational Fellowship of the Japan Society for the Promotion of Science (JSPS), the Humboldt Research Fellowship by the Alexander von Humboldt Foundation, the Hans Fischer Fellowship by the Institute of Advanced Study of the Technische Universität München, and the International Visiting Research Scholarship by the Peter Wall Institute of Advanced Study of the University of British Columbia. He was a recipient of the Best Paper Awards at the VLSI Test Symposium (VTS) in 2013 and IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems in 2015. He served as an ACM Distinguished Speaker for 2014–2020, a Distinguished Visitor for the IEEE Computer Society for 2013–2015, a Distinguished Lecturer for the IEEE Circuits and Systems Society for 2016–2017, the Chair for the IEEE Computer Society Tainan Chapter for 2013–2015, the Chair for the ACM SIGDA Taiwan Chapter for 2014–2015, the Secretary General for the IEEE Taipei Section for 2019–2020, and the VP Technical Activities for the IEEE CEDA for 2020–2023. Currently, he serves as the VP Conferences of IEEE CEDA, and the Executive Committee for ASP-DAC and ICCAD. He is a Distinguished Member of ACM.

Evangeline F. Y. Young (Fellow, IEEE) received the B.Sc. degree in computer science from The Chinese University of Hong Kong (CUHK), Hong Kong, and the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 1999.

She is currently a Professor with the Department of Computer Science and Engineering, CUHK. Her research interests include EDA, optimization, algorithms, and AI. Her research focuses on floor-planning, placement, routing, DFM, and EDA on physical design in general.

Dr. Young's research group has won championships and prizes in numerous renown EDA contests, including the CAD Contests at ICCAD, DAC, and ISPD. She has served on the organization committees for ICCAD and ISPD, and on the program committees for conferences, including DAC, ICCAD, ISPD, DATE, and ASP-DAC. She also served on the Editorial Boards for IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and *ACM Transactions on Design Automation of Electronic Systems, and Integration, the VLSI Journal*.