InstantGR: Scalable GPU Parallelization for Global Routing

Shiju Lin
sjlin@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Liang Xiao
lxiao23@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Jinwei Liu
jwliu@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Evangeline F.Y. Young
fyyoung@cse.cuhk.edu.hk
The Chinese University of Hong Kong

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 paper introduces useful techniques for parallelizing large-scale global routing that can significantly increase parallelism and thus reduce runtime. Building upon these techniques, we have developed an open-source GPU-based global router that achieves the state-of-the-art results in the latest ISPD'24 Contest benchmarks, thereby showcasing the effectiveness of our methods. The source code of this work is available at https://github.com/cuhk-eda/InstantGR.

1 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: global routing and 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 [15]. This estimation assists in generating physical-friendly netlists during logic synthesis and aids in partitioning, I/O planning, and timing budgeting during physical planning [15]. Given the purpose of estimation, global routers for early stages have the following three characteristics [15]. 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.

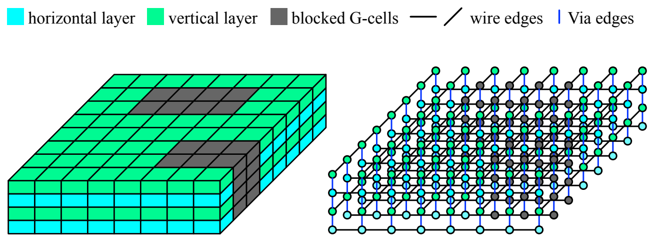
GPU computing has emerged as a promising solution to the runtime scalability issue due to its support of higher parallelism and memory bandwidth. Many GPU-accelerated algorithms have demonstrated significantly improved efficiency in various EDA problems [4–13, 16–21, 24–30]. Among these works, there are three related to global routing. GAMER [19] 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 [22]. FastGR [26] proposes a GPU-accelerated method for pattern routing and a heterogeneous task graph scheduler to improve speed. Lin and Wong [20] introduce a full-scale GPU-accelerated global router with multiple parallelization techniques, achieving more than 13× speedup over multithreaded CUGR [22].

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 are not yet explored. To overcome these problems, we propose new practical techniques to parallelize large-scale global routing. Our contributions are summarized as follows.

- We introduce a new method for batch generation. This method is based on 3D fine-grained overlap checking and explores more parallelism by increasing the number of nets per batch (nets in a batch can be routed in parallel).
- We also propose a node-level parallel routing approach that achieves much higher parallelism compared to traditional net-level parallel routing.
- Based on the above two techniques, we have developed a GPU-based scalable global router. Our global router outperforms the top-3 winners of the GPU/ML-Enhanced Large Scale Global Routing contest in the International Symposium

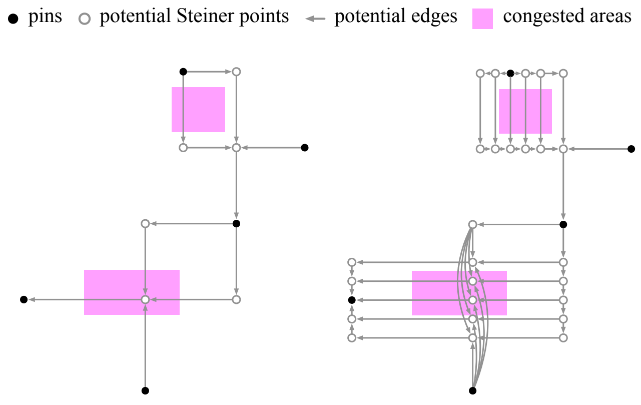
The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust. The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14218422).

ICCAD '24, October 27–31, 2024, New York, NY, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1077-3/24/10.
https://doi.org/10.1145/3676536.3676787



(a) G-cell partitioning. **(b) Grid graph.**

Figure 1: Global routing grid graph.



(a) Basic routing DAG. **(b) Augmented routing DAG.**

Figure 2: Global routing using routing DAG.

on Physical Design (ISPD) 2024 in both runtime and quality. The results demonstrate the effectiveness of our proposed techniques in practice.

The rest of the paper is organized as follows. We first present the preliminaries in Section 2. Next, we introduce two scalable techniques for parallelizing large-scale global routing in Section 3 and Section 4. Lastly, Section 5 shows the experimental results followed by the conclusions in Section 6.

2 PRELIMINARIES

2.1 Global Routing Formulation

In global routing, the multi-layer routing space is divided into a set of global cells (G-cells) (Figure 1a). 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 (Figure 1b). 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.

2.2 DAG-Based Global Routing

Our proposed parallel algorithm is mainly based on the DAG-based global routing algorithm in CUGR2 [23]. In DAG-based global routing, we can construct a routing DAG, i.e., directed acyclic graph, for each net to describe the set of 2D topologies we hope to explore. Figure 2a 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 3D 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 avoiding as illustrated in Figure 2b.

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. Firstly, the routing DAG limits the search space for each net and reduces racing conditions. Secondly, 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.

2.3 Evaluation Metrics

In the ISPD2024 contest, routing results are evaluated by a weighted sum of the total wirelength, via count, and overflow penalty [15]:

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

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 l-th layer is calculated as follows:

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

where s is a pre-defined 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 l-th layer, which is given. $OverflowScore$ is the summation of the overflow costs for all GCell edges.

3 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 3.1. Next, we will describe in Section 3.2 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 3.3.

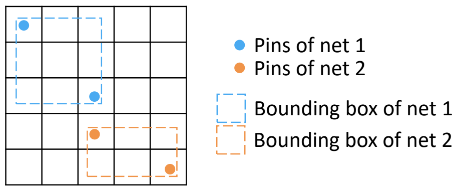
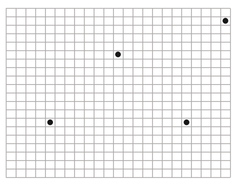
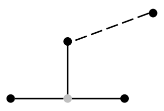


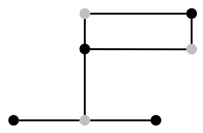
Figure 3: Example of net-level parallelism: if nets 1 and 2 are routed within their respective bounding boxes that do not overlap, they can be routed in parallel.

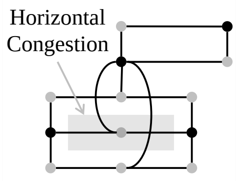
3.1 Net-Level Parallelism

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 Figure 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 [2, 3, 14, 19, 20, 22, 26], as most routing algorithms will restrict the routing graph for a net to enhance efficiency. For instance, Figures 4c and 4d show two example routing graphs of the 4-pin net depicted in Figure 4a (with RSMT in Figure 4b), both of which occupy only a small portion of the entire routing region (51 and 98 out of 480 G-cells in Figure 4c and Figure 4d respectively).


(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]

Figure 4: Two routing graph examples

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 non-overlapping nets that can be routed in parallel.

Batch Generation. Algorithm 1 shows a typical batch generation approach used by many routers [2, 3, 14, 19, 20]. 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.

Algorithm 1 A Typical Batch Generation Algorithm

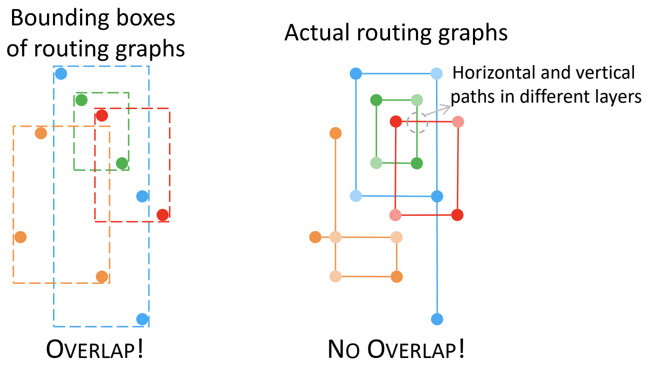
```
    Input: a set of nets $N$
    Output: batch count $m$, and batches of nets $B_i$ ($i = 1, \ldots, m$)
 1: $m \leftarrow 0$
 2: for net $n \in N$ do
 3:     for $batch\_id \leftarrow 1$ to $m$ do
 4:         if no overlap between $n$ and $B_{batch\_id}$ then
 5:             add net $n$ to batch $B_{batch\_id}$
 6:             break
 7:         end if
 8:     end for
 9:     if net $n$ has not been added to any batch then
10:         $m \leftarrow m + 1$
11:         create $B_m \leftarrow \{n\}$
12:     end if
13: end for
```

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 [19, 20, 22, 26] 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. Figure 5a shows four nets with non-overlapping L-shape routing graphs 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 1D 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.

3.2 Routing Graph Representation

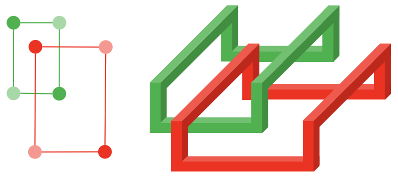
In the formulation of the ISPD2024 contest [15], the routing resources are all wires, and the use of both wires and vias consumes wire resources. Our representation of routing graphs uses horizontal and vertical segments. Such a representation can be obtained by converting all possible wires and vias of the routing graph into

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



Bounding boxes of routing graphs
Actual routing graphs
Horizontal and vertical paths in different layers
OVERLAP!
NO OVERLAP!

(a) For RSMT-based L-shape pattern routing, the routing graphs of these four nets do not overlap but their bounding boxes overlap.



(b) Three-dimensional 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).

Figure 5: Net-level parallelism for L-shape routing

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.*

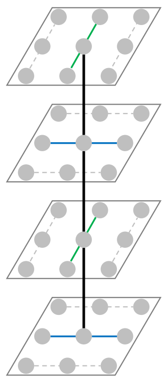
Definition 2. *(Routing Graph) A routing graph $G = (HS, VS)$ is represented by a set of horizontal segments HS and a set of vertical segments VS.*

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, 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.*

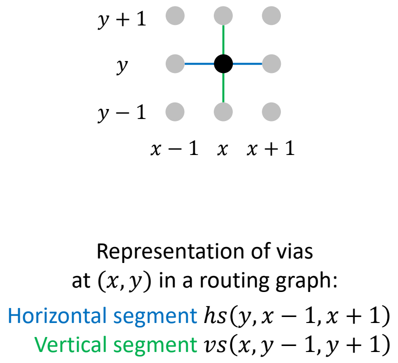
Definition 4. *(Routing Graph Overlap) Two routing graphs $G_0 = (HS_0, VS_0)$ and $G_1 = (HS_1, VS_1)$ overlap if there exist two overlapping horizontal segments $hs_0 \in HS_0$ and $hs_1 \in HS_1$ or two overlapping vertical segments $vs_0 \in VS_0$ and $vs_1 \in VS_1$.*

We will illustrate how to construct such a representation for a routing graph using Figure 4c as an example. For each of the wire segments in Figure 4c, we construct a corresponding horizontal or vertical segment. For each of the possible via locations (there are seven such locations in Figure 4c), we construct horizontal segments and vertical segments to account for its wire demands as shown in Figure 6, according to the via model described in [15]. These two simple steps complete the process of generating a representation for the L-shape routing graph in Figure 4c. The representation of any given routing graph can be generated in a similar fashion.

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 $x_{1_l} \le x \le x_{1_r}$.



(a) Modeling vias as wire demands [15]



$y+1$
y
$y-1$
$x-1$ x $x+1$
Representation of vias at (x, y) in a routing graph:
Horizontal segment $hs(y, x-1, x+1)$
Vertical segment $vs(x, y-1, y+1)$

(b) Segment representation of vias

Figure 6: Via representation by our routing graph segments.

3.3 Efficient Overlap Checking Algorithms

In this subsection, 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 one-dimensional 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).

- **Insertion.** Insert a segment $s = [x_l, x_r]$ to S.
- **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 [1] 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 [15], 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.

Algorithm 2 Point Exhaustion

```
 1: initialize $[b_1, ..., b_n] = [0, ..., 0]$
 2: procedure INSERT($s$)                          ▷ segment $s = [l, r]$
 3:     for $i \leftarrow l$ to $r$ do
 4:         $b_i \leftarrow 1$
 5:     end for
 6: end procedure
 7: procedure QUERY($s$)                           ▷ segment $s = [l, r]$
 8:     for $i \leftarrow l$ 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
```

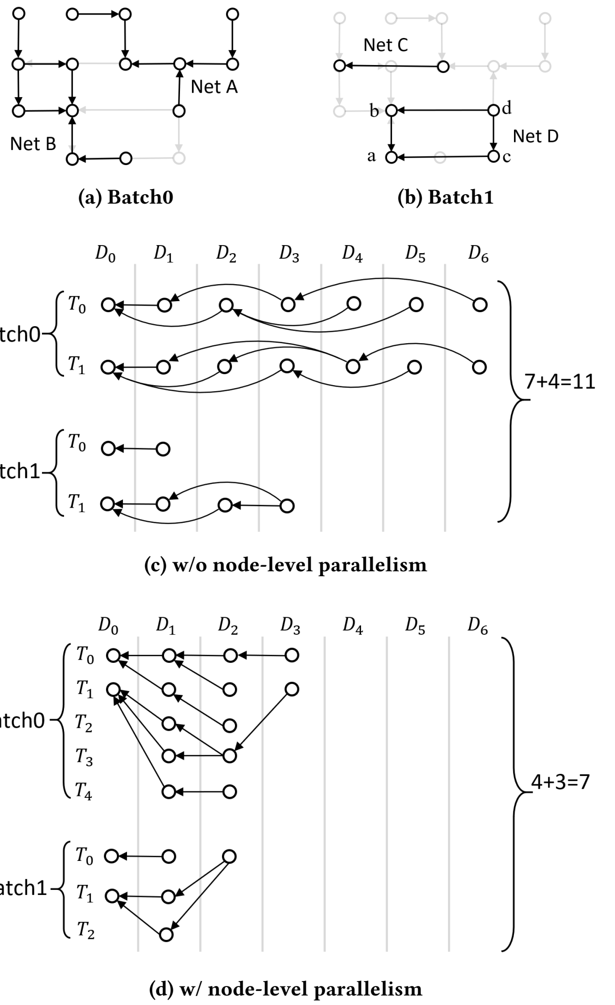
We can further improve the efficiency of this point exhaustion by using bit arrays, which are effective at exploiting bit-level parallelism in hardware to perform operations quickly [31] (also known as bitsets). Take Algorithm 2 with $n = 64$ as an example. We can use one 64-bit integer i to store the Boolean values of $[b_1, ..., b_{64}]$. 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 5.1.

Observation 2: more queries than insertions. As shown in Algorithm 1, each net needs multiple queries (line 4) until a non-overlapping 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× 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 non-exact 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.

4 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 Figure 7b). 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 incoming nodes. By routing nodes of the same depth in parallel, we can achieve node-level parallelism.



Net A
Net B
Net C
Net D
b
d
a
c
(a) Batch0
(b) Batch1
D_0 D_1 D_2 D_3 D_4 D_5 D_6
Batch0
T_0
T_1
Batch1
T_0
T_1
7+4=11
(c) w/o node-level parallelism
D_0 D_1 D_2 D_3 D_4 D_5 D_6
Batch0
T_0
T_1
T_2
T_3
T_4
Batch1
T_0
T_1
T_2
4+3=7
(d) w/ node-level parallelism

Figure 7: An example of node-level parallelism

Figure 7 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 Figure 7a, and nets C and D are distributed to batch 1. Figure 7c 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 Figure 7c, by routing nodes with the same depth simultaneously, we only need 4 steps to

Table 1: Benchmark Details [15].

	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

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 less 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 depth of maximum depth of all its outgoing nodes.

5 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 the ISPD2024 contest environment [15], all the experiments were run with 4 NVIDIA A800 GPUs and 8 CPU threads. We used the same benchmarks (details shown in Table 1) and evaluator as in the contest [15].

5.1 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 2. 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 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

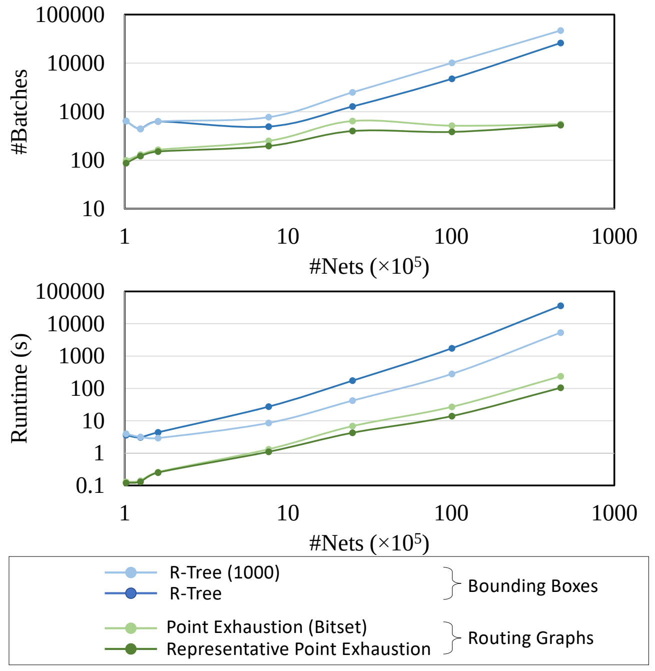


Figure 8: Scalability comparisons of different overlap checking methods (experiments on benchmarks 7–13)

Table 2, which is significantly smaller than that of using R-trees for bounding boxes or using segment trees for our representation.

We present the batch generation results of benchmarks 7–13 in Figure 8 to demonstrate the scalability of our approach. When the number of nets increases, our method increases slower 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 integrated circuits keep increasing in size and complexity.

5.2 Global Routing

We conduct our experiments on the ISPD2024 contest benchmark suites[15] and evaluate the results using the official evaluator of the contest. For a fair comparison, we obtained the top 3 teams' binary files and evaluated them on our machine using their default settings.

Compared to the first-place team, our router produces an average of 100.55% wirelength and 98.83% overflows, and with significantly fewer vias (82.54%). The comparison of summed score on all cases are shown in Table 3. In the table, the "Score" is given by the evaluator, which is computed using Equation (1). Our router achieves the best total scores in all cases, with a 2.01× acceleration compared to the first-place team, demonstrating the significant advantage of our novel batch generation algorithm and node-level parallelism.

5.3 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

Table 2: Batch Generation Results with Different Overlap Checking Methods on Two Largest Benchmarks

Geometry	Method	Benchmark 12		Benchmark 13		Overlap
		#Batches	Time (s)	#Batches	Time (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	127	554	1101	No
	Point Exhaustion	515	35	554	297	No
	Point Exhaustion (Bitset)	515	27	554	239	No
	Representative Point Exhaustion	383	14	527	105	2% Length

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

Table 3: Experimental Results of Top-3 Global Routers of ISPD2024 Contest and InstantGR

Benchmark	1st Place		2nd Place		3rd Place		InstantGR	
	Score	Time (s)	Score	Time (s)	Score	Time (s)	Score	Time (s)
0	19789143	3.44	20099496	2.10	19897356	1.62	**19716744**	2.55
1	15241650	2.95	15432389	1.97	15280353	1.86	**15126510**	2.56
2	48257027	5.60	48837685	3.31	48257010	3.87	**47982224**	3.73
3	113562198	8.20	113321049	18.49	112592863	18.25	**112474880**	14.64
4	398169317	56.59	411758419	35.53	403915333	27.21	**397658013**	35.78
5	1626227314	231.87	1665748885	142.16	1639553927	200.70	**1623946297**	116.63
6	19609525592	2821.53	19684092627	2115.14	19730043753	2748.29	**19139291553**	1289.02
7	22602876	3.75	23093501	2.81	22821289	1.72	**22545800**	2.98
8	13827142	2.49	14133269	2.45	13867124	1.46	**13774789**	2.87
9	43195979	4.97	44141552	4.29	43295092	4.14	**43047147**	3.83
10	113109620	16.71	110986781	15.87	111778586	9.71	**109844055**	9.45
11	383637652	43.75	395488859	40.20	388200338	29.42	**382639889**	36.35
12	1782191834	214.00	1824527054	148.90	1795524941	229.85	**1780897390**	117.75
13	12528609838	1654.27	12676067016	1338.30	12597498026	2181.39	**12257767067**	882.66
Average	2622710513	362.15	2646266327	276.54	2638751857	389.96	2569050883	180.06
Ratio	1.02	2.01	1.03	1.54	1.03	2.17	**1.00**	**1.00**

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 4.

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 4, where we list the sum of node counts for the bottleneck nets in all batches. In contrast, in our node-level parallel approach, the bottleneck for each batch comes from the net with the largest depth, which is significantly less (14.3×) than the number of nodes. The reason for the substantial ratio of node to depth in Table 4 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 4. 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.

6 CONCLUSIONS

In this work, we have achieved a breakthrough compared to the traditional bounding box-based overlap checking methods, developing a precise and efficient batch generation algorithm. Our algorithm significantly increases the number of nets per batch for parallel routing, thereby fully unlocking the advantage of GPUs over CPUs. This algorithm has the potential to be applied in many other routers that involve net-level parallelism. Besides, we also propose a node-level parallel algorithm to process nodes with the same depth, which further accelerated our program. Experiments on ISPD 2024 contest benchmark show that our algorithm can achieve 2.1% improvement in quality and 2.01× acceleration compared to the first place.

REFERENCES

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

Table 4: 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.

[2] Gengjie Chen, Chak-Wa Pui, Haocheng Li, Jingsong Chen, Bentian Jiang, and Evangeline F. Y. Young. 2019. Detailed routing by sparse grid graph and minimum-area-captured path search. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '19)*. Association for Computing Machinery, New York, NY, USA, 754–760.

[3] Gengjie Chen, Chak-Wa Pui, Haocheng Li, and Evangeline F. Y. Young. 2020. Dr. CU: Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 9 (2020), 1902–1915. https://doi.org/10.1109/TCAD.2019.2927542

[4] Guojin Chen, Haoyu Yang, and Bei Yu. 2023. GPU-accelerated matrix cover algorithm for multiple patterning layout decomposition. In *DTCO and Computational Patterning II*, Ryoung-Han Kim and Neal V. Lafferty (Eds.), Vol. 12495. International Society for Optics and Photonics, SPIE, 124951P.

[5] Guannan Guo, Tsung-Wei Huang, Yibo Lin, Zizheng Guo, Sushma Yellapragada, and Martin D. F. Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 11 (2023), 4219–4232.

[6] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[7] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 721–726. https://doi.org/10.1109/DAC18074.2021.9586316

[8] Zizheng Guo, Feng Gu, and Yibo Lin. 2022. GPU-Accelerated Rectilinear Steiner Tree Generation. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 53, 9 pages.

[9] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated static timing analysis. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 147, 9 pages.

[10] Zizheng Guo, Jing Mai, and Yibo Lin. 2021. Ultrafast CPU/GPU Kernels for Density Accumulation in Placement. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1123–1128. https://doi.org/10.1109/DAC18074.2021.9586149

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

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

[13] Andrew B. Kahng and Zhiang Wang. 2024. DG-RePlAce: A Dataflow-Driven GPU-Accelerated Analytical Global Placement Framework for Machine Learning Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3436521

[14] Haocheng Li, Gengjie Chen, Bentian Jiang, Jingsong Chen, and Evangeline F. Y. Young. 2019. Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7.

[15] Rongjian Liang, Anthony Agnesina, Wen-Hao Liu, and Haoxing Ren. 2024. GPU/ML-Enhanced Large Scale Global Routing Contest. In *Proceedings of the 2024 International Symposium on Physical Design (ISPD '24)*. Association for Computing Machinery, New York, NY, USA, 269–274.

[16] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Brucek Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247942

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

[18] Shiju Lin, Jinwei Liu, Tianji Liu, Martin D. F. Wong, and Evangeline F. Y. Young. 2022. NovelRewrite: node-level parallel AIG rewriting. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 427–432. https://doi.org/10.1145/3489517.3530462

[19] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2023. GAMER: GPU-Accelerated Maze Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2023), 583–593. https://doi.org/10.1109/TCAD.2022.3184281

[20] Shiju Lin and Martin D. F. Wong. 2022. Superfast Full-Scale GPU-Accelerated Global Routing. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22)*. Article 51, 8 pages. https://doi.org/10.1145/3508352.3549474

[21] Yibo Lin, Wuxi Li, Jiaqi Gu, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2020. ABCDPlace: Accelerated Batch-Based Concurrent Detailed Placement on Multithreaded CPUs and GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 5083–5096. https://doi.org/10.1109/TCAD.2020.2971531

[22] Jinwei Liu, Chak-Wa Pui, Fangzhou Wang, and Evangeline F. Y. Young. 2020. CUGR: Detailed-Routability-Driven 3D Global Routing with Probabilistic Resource Model. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218646

[23] Jinwei Liu and Evangeline F.Y. Young. 2023. EDGE: Efficient DAG-based Global Routing Engine. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[24] Lixin Liu, Bangqi Fu, Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2024. Xplace: An Extremely Fast and Extensible Placement Framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 6 (2024), 1872–1885. https://doi.org/10.1109/TCAD.2023.3346291

[25] Lixin Liu, Bangqi Fu, Martin D. F. Wong, and Evangeline F. Y. Young. 2022. Xplace: an extremely fast and extensible global placement framework. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1309–1314. https://doi.org/10.1145/3489517.3530485

[26] Siting Liu, Yuan Pu, Peiyu Liao, Hongzhong Wu, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, and Bei Yu. 2023. FastGR: Global Routing on CPU–GPU With Heterogeneous Task Graph Scheduler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 7 (2023), 2317–2330. https://doi.org/10.1109/TCAD.2022.3217668

[27] Tianji Liu, Lei Chen, Xing Li, Mingxuan Yuan, and Evangeline F. Y. Young. 2024. FineMap: A Fine-Grained GPU-Parallel LUT Mapping Engine. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference* (Incheon, Republic of Korea) *(ASPDAC '24)*. IEEE Press, 392–397.

[28] Tianji Liu, Yang Sun, Lei Chen, Xing Li, Mingxuan Yuan, and Evangeline F.Y. Young. 2024. A Unified Parallel Framework for LUT Mapping and Logic Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3429079

[29] Tianji Liu and Evangeline F.Y. Young. 2023. Rethinking AIG Resynthesis in Parallel. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247961

[30] Yang Sun, Tianji Liu, Martin D.F. Wong, and Evangeline F.Y. Young. 2024. Massively Parallel AIG Resubstitution. In *2024 61st ACM/IEEE Design Automation Conference (DAC)*.

[31] Wikipedia. 2024. Bit array — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bit_array. [Online; accessed 29-April-2024].