FastGR: Global Routing on CPU–GPU With Heterogeneous Task Graph Scheduler

Siting Liu, Yuan Pu, Peiyu Liao, Hongzhong Wu, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, *Member, IEEE*, and Bei Yu, *Senior Member, IEEE*

***Abstract*—Running time is a key metric across the standard physical design flow stages. However, with the rapid growth in design sizes, routing runtime has become the runtime bottleneck in the physical design flow. As a result, speeding routing becomes a critical and pressing task for IC design automation. Aside from the running time, we need to evaluate the quality of the global routing solution since a poor global routing engine degrades the solution performance after the entire routing stage. This work takes both of them into consideration. We propose a global routing framework with GPU-accelerated routing algorithms and a heterogeneous task graph scheduler, called FastGR, to accelerate the procedure of the modern global router and improve its effectiveness. Its runtime-oriented version FastGR^{L} achieves 2.489× speedup compared with the state-of-the-art global router. Furthermore, the GPU-accelerated L-shape pattern routing algorithm used in FastGR^{L} can contribute to 9.324× speedup over the sequential algorithm on CPU. Its quality-oriented version FastGR^{H} offers a 27.855% improvement of the number of shorts over the runtime-oriented version and still gets 1.970× faster than the most advanced global router.**

***Index Terms*—Parallel algorithms, routing.**

I. Introduction

ROUTING is an essential stage in the design flow of the modern very-large-scale integration (VLSI). Global routing and detailed routing are two stages of the modern routing flow. Global routing produces routing guidance for detailed routing by performing rough routing on a coarse grid graph [1], [2], [3]. Following the guide from global routing, detailed routing performs on a fine grid graph to interconnect all the wires and eliminate design rule violations [4]. Global routing also functions as a congestion predictor for other phases in the design cycle, such as placement [5], [6]. The efficiency and efficacy of global routing are crucial to the design closure due to its recurrent invocation and guiding role.

Determining the shortest connections for each net is a critical problem in global routing [7]. Due to the enormous problem scale, the modern global router is always divided into two stages: 1) the general routing stage and 2) the rip-up and reroute iterations. To narrow the search space for efficiency, the pattern routing algorithm is always used in the general routing stage [8]. To obtain higher solution performance, the rip-up and reroute iterations always use maze routing by doing an extended search to discover paths for all the nets, which cannot find a legal paths in the general routing stage.

Due to the significance of the global routing step, various efforts have been made to improve both the solution quality and the efficiency. Better global routing solutions, on the other hand, usually result in longer searching time since more candidate routing paths are explored. Existing global routing approaches primarily focus on improving CPU efficiency [9], [10], [11], [12], whereas the speedup is limited owing to threading overhead, limit bandwidth, and CPU cache sizes. Meanwhile, GPUs have a large number of grid-based processing resources and small synchronization costs inside the computation blocks. With GPU power rising all the time, speeding global routing on heterogeneous CPU–GPU platforms opens up new possibilities for high-performance routing engines.

The literature has extensively explored shortest path searching with GPU [13], [14]. However, most work only explores the basic single-source shortest path algorithm. The task only needs to search for one shortest path on a large graph. These algorithms are unsuitable for routing since we must route millions of nets while considering numerous objectives and limitations, such as wirelength, number of vias, and design rules. Regarding those modern routing challenges, more appropriate GPU kernel algorithms should be designed. In this work, we propose FastGR, a global routing framework accelerated for CPU–GPU platforms. The framework leverages a GPU-friendly pattern routing algorithm and a task graph scheduler for heterogeneous CPU–GPU systems. By utilizing the processing resources of GPUs, we can further increase the solution quality performance of our global routing framework

Manuscript received 16 April 2022; revised 10 August 2022; accepted 14 October 2022. Date of publication 27 October 2022; date of current version 20 June 2023. This work was supported in part by The Research Grants Council of Hong Kong SAR under Project CUHK24209017, and in part by the National Science Foundation of China under Project 62141404 and Project 62034007. The preliminary version has been presented at the IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE) in 2022 [DOI: 10.23919/DATE54114.2022.9774606]. This article was recommended by Associate Editor L. Behjat. *(Corresponding authors: Bei Yu; Yibo Lin.)*

Siting Liu and Peiyu Liao are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, SAR, and also with the School of Integrated Circuits, Peking University, Beijing 100871, China.

Yuan Pu and Bei Yu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, SAR (e-mail: byu@cse.cuhk.edu.hk).

Hongzhong Wu and Rui Zhang are with HiSilicon, Shenzhen 518129, China.

Zhitang Chen is with Huawei Noah's Ark Lab, Hong Kong.

Wenlong Lv is with Huawei Technologies Company, Shenzhen, China.

Yibo Lin is with the School of Integrated Circuits, Peking University, Beijing 100871, China, and also with the Beijing Advanced Innovation Center for Integrated Circuits, Beijing 100871, China (e-mail: yibolin@pku.edu.cn).

Digital Object Identifier 10.1109/TCAD.2022.3217668

while incurring a little runtime overhead. We develop two variants of our global routing framework: the runtime-oriented version FastGR$^{\mathrm{L}}$ and the quality-oriented version FastGR$^{\mathrm{H}}$.

The major contributions of this work are summarized as follows.

1) We propose a novel GPU-friendly pattern routing framework that can route a batch of nets while taking use of the massive parallelism in routing problem on GPU.
2) We present a GPU-accelerated L-shape pattern routing technique and an innovative GPU-accelerated hybrid pattern routing algorithm by reformulating them into computation graph flows.
3) We present an effective task graph scheduler for distributing tasks on CPU–GPU systems considering workload balancing.
4) Experiments show that when compared to the state-of-the-art global router [3], our runtime-oriented version FastGR$^{\mathrm{L}}$ can achieve 2.489× overall speedup without any quality degradation. In particular, the GPU-accelerated L-shape pattern routing algorithm can bring 9.324× speedup in pattern routing; Meanwhile, the task scheduler can bring 2.070× speedup in the rip-up and reroute stage.
5) The quality-oriented version FastGR$^{\mathrm{H}}$ reduces the number of shorts by 27.855% over the runtime-oriented version FastGR$^{\mathrm{L}}$ [15] while remaining 1.970× faster than the most advanced global router [3].

The remainder of this article is structured as follows. Section II discusses the problem definition, the background of modern global routing algorithms. Section III describes our GPU-friendly pattern routing algorithms and the efficient task graph scheduler. Section IV validates the algorithms with experimental results. In the end, Section V concludes this article.

II. Preliminaries

A. Problem Formulation

Global routing works on a collection of global routing cells (G-cells), which essentially form horizontal and vertical grids distributed uniformly. A grid graph $G(V, E)$ is defined to formulate a global routing problem by considering each G-cell as a vertex ($v \in V$) and drawing an edge ($e \in E$) between all the pairs of adjacent G-cells. The wire edge is the edge between two G-cells on the same metal layer. Its capacity is equal to the number of tracks that can be provided for all the wires, while its demand is the number of tracks that all the wires need to go through. The via edge is the edge between two G-cells with the same 2-D position but on separate metal layers. Many 2-D global routers set the via capacity as infinite to ignore the cost of vias, while some 3-D global routers consider the via capacity, e.g., CUGR [3].

Fig. 1 illustrates the procedure of grid graph construction. We map all the pins into G-cells according to the pin position. In this sample, different colors represent different metal layers. There is a preferred routing direction (horizontal or vertical) for wire edges in each metal layer, represented as the colored solid lines. The black dotted lines mean the via edges in our grid graph.

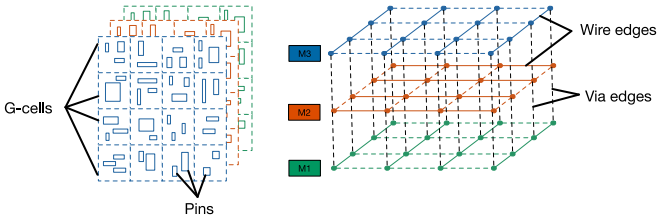


Fig. 1. Grid graph construction procedure; there are three metal layers with 4×4 grids in each layer. The final grid graph is shown in the right.

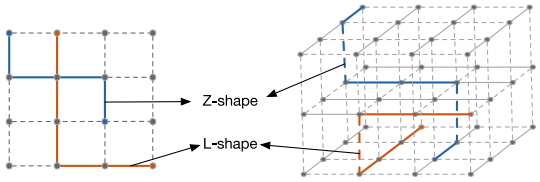


Fig. 2. 2-D/3-D pattern routing; the red path represents one L-shape pattern routing solution, and the blue path is one of the candidate Z-shape pattern routing paths.

Modern global routers always propose different cost functions for each edge to consider the grid edges' wirelength, congestion, and net delay. With the grid graph G construction, the global routing problem can be formulated as the minimum accumulated cost path searching problem on G for all the nets defined in VLSI designs.

B. Modern Global Router

A multipin net n to be routed includes a set of points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ on G, where the pin is the single point of a net. In the modern global router, the multipin nets are always transformed into a collection of two-pin nets first with various Steiner tree construction techniques [16], [17], [18], [19]. The Steiner tree construction can help to lead researchers to the well-developed area of single-source single-sink shortest path searching. Pattern routing [8], [20] plays an important role in the modern global routing framework because of its efficiency. Two popular patterns are shown in Fig. 2. We illustrate the L-shape and Z-shape pattern routing paths on 2-D and 3-D routing spaces. As shown in Fig. 2, the L-shape pattern routing path includes one single bend point to change the routing direction, while the Z-shape pattern routing path contains two bend points.

The most basic way of routing is to choose a certain nets order and then route these nets consecutively in that order. However, the key drawback of such a sequential method is that it may suffer from the net ordering strategy and result in a poor routing solution since the previously routed net might impede the routing for its succeeding nets. Different net ordering strategies will bring different influences to the final global routing solution, which we will discussed in Section IV-C. Modern sequential global routers always follow a two-stage process [21], [22], [23], the general routing stage and rip-up and reroute iterations, with a net-ordering scheme. The most

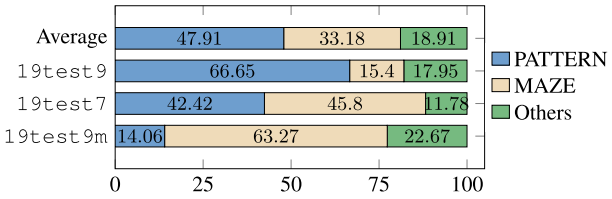


Fig. 3. Runtime breakdown of a typical global router, CUGR. PATTERN represents the runtime taken by the pattern routing stage; MAZE represents the runtime taken by the rip-up and reroute stage.

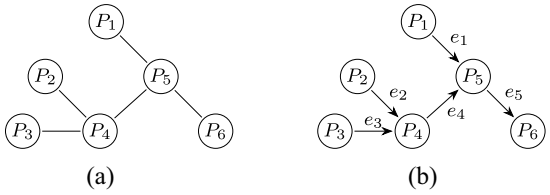


Fig. 4. Example of intranet ordering. (a) Multiple two-pin nets. (b) Two-pin nets with order.

common-used framework applies pattern routing for the general routing stage and maze routing for the rip-up and reroute iteration. Our framework also follows this two-stage procedure and named the general routing stage as the pattern routing stage directly.

The concurrent routing approaches can solve the problem related to net ordering and route all of nets at once. The most often-used concurrent technique is to model the global routing problem as 0–1 integer linear programming problem (0–1 ILP) [24], [25]. Despite the fact that such an ILP formulation can discover the optimal solution when it exists, the 0–1 ILP problem is an NP-complete problem. The high temporal complexity constrains the possible problem size, which is unacceptable in the industry.

So as to have an efficient framework, we develop a routing algorithm with a practical routing task graph scheduler. There is an efficient two-stage sequential routing structure, encompassing pattern routing, and maze routing. After pattern routing, maze routing is adopted in the rip-up and reroute iterations to achieve routing closure. In such situation, the pattern routing stage route roughly twice as many nets as the first rip-up and reroute iteration.

C. Runtime Breakdown

We demonstrate the runtime breakdown of a modern global router. It consists of two stages: 1) a pattern routing stage and a rip-up and 2) reroute stage with maze routing. The runtime breakdown of the global router on three benchmarks from the ICCAD2019 benchmark suit [26] are plotted in Fig. 3. PATTERN denotes the runtime portion of the pattern routing stage, whereas MAZE represents the runtime portion of the maze routing algorithm for rip-up and reroute iterations. As shown in Fig. 3, `19test9` is a PATTERN-dominated, `19test9m` is an MAZE-dominated design, and `19test7` is a design with approximately the same proportion of PATTERN and MAZE. Fig. 3 demonstrates that it is PATTERN-dominated on average because the number of nets that the pattern routing stage process is substantially more than the number of nets that the rip-up and reroute iterations should handle.

D. Intranet Ordering

As mentioned before, it is a typical practice to break a multipin net in multiple two-pin nets in sequential global routing. We are expected to establish the net ordering of these two-pin nets since there is a dependence between each pair of connected two-pin nets in our dynamic programming-based algorithms. One of the most common approaches is to use a depth-first search (DFS) traversal to explore all nodes starting from a random root. Take Fig. 4 as an example to demonstrate this procedure. All of the two-pin nets will route in the reverse order sequentially.

Assume we select P_6 as the random root in Fig. 4(a). Then, beginning with P_6, we conduct DFS traversal. The DFS traversal accesses all the nodes in the following order: P_6, P_5, P_4, P_3, P_2, and P_1. As illustrated in Fig. 4(b), we mark all the two-pin nets in the reverse sequence e_1, e_2, e_3, e_4, and e_5, which is the order in which the routing algorithm will be performed.

E. Internet Ordering

Besides the net ordering strategy within a single multipin net, we also need to consider the ordering of routing between multipin nets. Net ordering has a substantial influence on routing solution quality since a net routed early may hamper the nets routed later with fewer routing resources [27], [28]. Therefore, efficient Internet ordering techniques are desirable in pattern routing.

Unfortunately, finding a universally optimal ordering scheme is extremely difficult. The literature indicates that no single Internet ordering technique can outperform others in all the benchmarks [29]. Typical Internet ordering schemes include: 1) sort the nets according to the number of pins in ascending (descending) order, as nets with more pins can be more likely to block the routing of other nets; 2) sort according to the wirelength of nets, as shorter nets are not as flexible as longer nets and routing shorter ones first can improve routability; and 3) sort according to the bounding box area of nets, as larger nets require more routing resources and thus they should be routed first.

Typical global routing algorithms adopt the aforementioned Internet ordering strategies. However, such strategies follow the sequential nature of net-by-net routing, causing the challenges in efficiency. Hence, in this work, we explore a heterogeneous task graph scheduler for routing nets considering both parallelization and workload balancing on CPU–GPU platforms.

III. Algorithms

A. Overview

Fig. 5 depicts the overall flow of FastGR. To begin, we present a heterogeneous task graph scheduler and use it to manage the execution order of multiple routing tasks in both

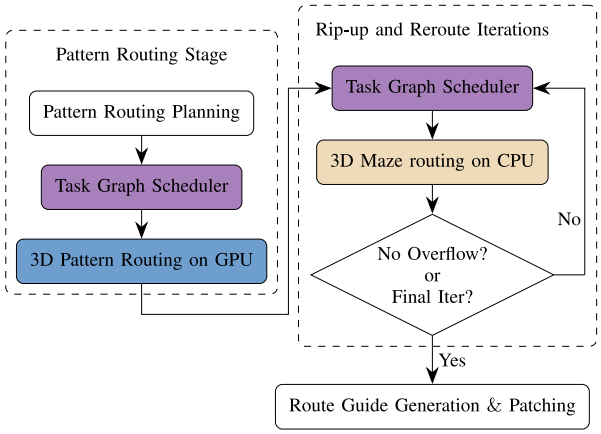


Fig. 5. Overall flow of FastGR.

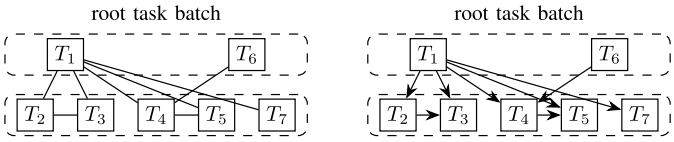


Fig. 6. Sample for task graph scheduler with seven tasks; the edge in the task graph represents the conflict relationship for each pair of connected tasks.

Algorithm 1 Batch Extraction Algorithm

Require: `nets`: the set of nets which need to process the pattern routing.
Ensure: `batch`: a set of nets batches.
1: $e \leftarrow$ `nets`[0];
2: Remove e from `nets` and declare a new empty batch `batch` $\leftarrow \{e\}$;
3: **for** $e_i \in$ `nets` **do**
4: **if** e_i has no conflict with all the nets in `batch` **then**
5: Push e_i into `batch`;
6: Remove e_i from `nets`;
7: **end if**
8: **end for**
9: **return** `batch`;

portions of our global routing framework, the pattern routing stage, and the rip-up and reroute iterations. The conflicting relationship among these tasks is used to form the task graph. It is important to note that a conflict between two routing tasks indicates that they cannot be processed at the same time. Our task graph scheduler is utilized to determine the execution order of each conflicting pair of tasks.

The pattern routing planning stage contains the Steiner tree construction, the edge shifting algorithm to optimize the Steiner tree, and a scheduler to get the routing order for the two-pin nets in the pattern routing stage. After determining the execution order of the task graph using our task graph scheduler, we employ our proposed 3-D GPU-friendly pattern routing algorithm on GPU.

For each iteration in the rip-up and reroute stage, we first extract the nets to rip up and consider each net as a rip-up routing task. We can maximize the utilization of parallelism across of these rip-up routing tasks using our task graph scheduler. Then, on the CPU, we perform a 3-D maze routing algorithm to complete the reroute iteration. We generate routing guidance and patches for the detailed routing after multiple rip-up and reroute iterations.

In the following sections, we will go over the details of our task graph scheduler, our proposed GPU-friendly pattern routing framework, our GPU-friendly 3-D L-shape pattern routing algorithm, our GPU-friendly 3-D hybrid-shape pattern routing algorithm, and our task graph scheduler techniques in both stages.

B. Task Graph Scheduler

To determine the execution order of the global routing tasks, we develop a two-stage task graph scheduler. The first stage is to create a task conflict graph based on the conflicting relationship between each pair of tasks. The task graph scheduler is then used to establish the order of execution for each conflict edge in the task conflict graph.

Following the task conflict graph generation, we extract one root task batch based on the conflict information in the graph. Since there is no conflicts inside the root task batch, all of these tasks can be divided into two groups: 1) the root task batch and 2) the nonroot task batch. There are only two situations between each pair of conflicting tasks since the no-conflict situation inside the root task batch.

1) *One task is part of the root task batch, whereas the other is not.* The execution direction is from the root batch task to the other.
2) *Both the tasks are not part of the root batch.* The execution order is from the task with a smaller task ID to the other and the task ID indicates the sorting result.

Our task graph scheduler uses the above strategy to assign the execution order to each pair of conflicting tasks. As an example in Fig. 6, we first select an independent root task batch from the task conflict graph. Then, using the above assignment strategy, we can obtain the final execution order for the task conflict graph.

C. Pattern Routing Stage: Task Graph Generation

In the pattern routing stage, we consider a batch of multipin nets as a single routing task because the number of nets to be routed in this stage is quite enormous. To take use of the parallelism across all of the multipin nets, we first partition the multipin nets into multiple batches using a batch extraction technique based on [4], as defined in Algorithm 1, to maximize the parallelism within each batch.

Given a set of multipin nets `nets`, we first sort all of the nets using a sorting strategy described in Section IV-C. Assume we sort all of the nets with increasing bounding box areas. Then, in one new empty batch `batch`, select the first net e (Line 1), the net with the smallest bounding box area in the collection of remaining nets. Following that, we scan the whole collection of nets in sequence and filter out the nets that do not conflict with any of the nets in `batch` (lines 3–8). We update the list of remaining nets `nets` and the new batch `batch` whenever we identify one net that fulfill this no-conflict requirement. After such a thorough scan, we generate

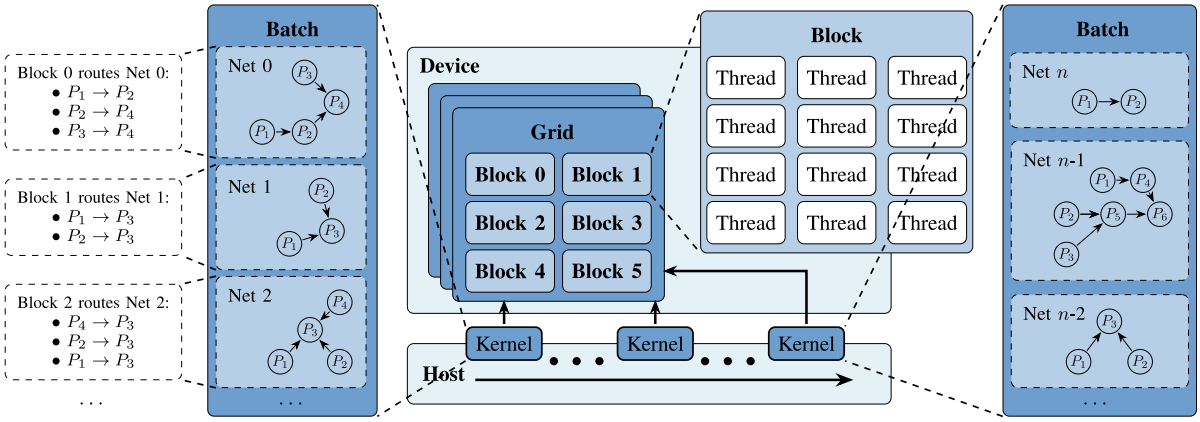


Fig. 7. Programming architecture for the pattern routing stage; on the host, the kernels are invoked sequentially and each of them is used to process a single batch of nets. The nets in the same batch will be routed simultaneously on different blocks on the device.

a batch `batch` with nearly optimal independent sets of nets. We should repeat the batch scheduler until the set of remaining nets is empty. Finally, with a little overhead, we can acquire a collection of no-conflict nets batches.

Since there is no bounding box overlap inside each batch, we can route the nets in the same batch at the same time, allowing us to treat one batch as one routing task when constructing the task graph in the pattern routing stage. The task graph we generate from these batches will be a complete graph with edges between every two batches, according to the batch extraction technique described in Algorithm 1. To prevent execution conflicts, we will execute all these routing tasks sequentially by using our task graph scheduler.

Fig. 7 shows the programming architecture of our GPU-friendly pattern routing framework for all of these routing tasks during the pattern routing stage. Each batch in Fig. 7 represents a single routing task, and each task contains several multipin nets.

To accomplish this single routing task, we invoke the 3-D pattern routing kernel on the host, as demonstrated in Fig. 7. We will allocate each kernel to the device's grid, and multiple blocks in this grid will be used concurrently with distinct computation flows. The separate blocks can manage the pattern routing procedure for various multipin nets simultaneously. Furthermore, all threads in a single block can conduct the same calculation flow at the same time. As a result, formulating the pattern routing procedures in each two-pin net into a uniform computation flow is good for the utilization of GPUs.

D. Pattern Routing Stage: GPU-Friendly L-Shape Pattern Routing

Besides the parallelism among multipin nets discussed above, there is also the possibility of parallelism within each two-pin net. As shown in Fig. 7, We use multiple blocks on GPU to execute the multipin nets in the same batch simultaneously in one-to-one correspondence. Furthermore, for each multipin net in the block, several two-pin nets should be routed in an order decided by DFS traversal, which we have discussed in Section II-D. Having the ordered multipin net, we apply the bottom-up dynamic programming to solve the

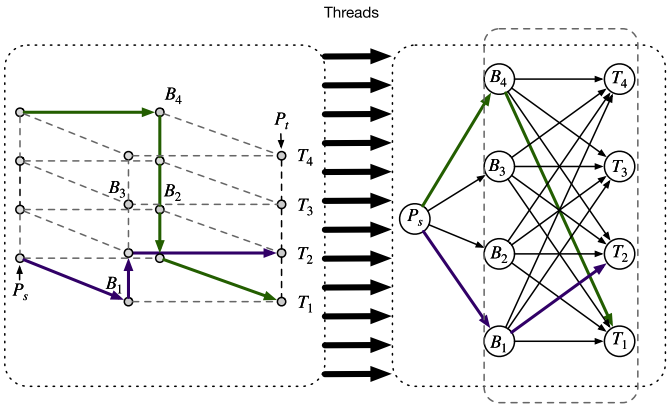


Fig. 8. GPU-friendly 3-D L-shape pattern routing; each 3-D L-shape pattern routing solution is denoted as $\mathcal{P}\{P_s, B_{l_s}, T_{l_t}\}$. There are two sample paths colored by green and purple.

3-D global routing problem of each multipin net [3]. Pattern routing is a widely used method to solve the global routing problem for the single two-pin net. To utilize the computation power of homogeneous GPU threads, we reformulate the conventional 3-D L-shape pattern routing algorithm into a unified computation graph flow, which is demonstrated in Fig. 8.

Referring to the cost scheme in [3], $c_{\mathrm{w}}(u, v, l)$ is used to represent *the cost of a wire edge*, considering wirelength cost and congestion cost, where l is the metal layer of the edge; u and v are two 2-D G-cells connected by this edge. Meanwhile, $c_{\mathrm{v}}(u, l_1, l_2)$ is applied to mean *the cost of a via edge*, where l_1 and l_2 are metal layers connected by the via with the same 2-D position to the G-cell u. Table I defines some important notations used in our GPU-friendly 3-D pattern routing algorithms for the two-pin net $P_s \rightarrow P_t$.

Take a two-pin net $P_s \rightarrow P_t$ as the example. Suppose that the number of the metal layers is 4. The left part in Fig. 8 shows two separate 3-D L-shape pattern routing solutions of $P_s \rightarrow P_t$ in different colors. We use l_s to represent the source layer of the wire connecting the source point to the bend point, while l_t is the target layer of the wire connecting the bend point

TABLE I
NOTATIONS FOR GPU-FRIENDLY PATTERN ROUTING

Notation	Description
L	The number of metal layers.
B	The bend point of 2D L-shape patterns
B_l	The bend points on l^{th} layer of 3D L-shape patterns.
T_l	The point on l^{th} layer with the same 2D position as P_t.
$\mathcal{T}(P_s)$	The sub-tree rooted at P_s in the ordered multi-pin net.
$c(\cdots)$	The cost of a routing path for a single two-pin net.
$c_{\text{bc}}(P_s, l_s)$	The bottom children cost with P_s in the l_s layer.
$c^*(P_s, P_t, l_t)$	The minimum cost of $\mathcal{T}(P_s)$ attached with $P_s \rightarrow T_{l_t}$.
$\boldsymbol{w}^{(i)}$	Weight vector of part i in our computation graph flow.
$\mathbf{W}^{(i)}$	Weight matrix of part i in our computation graph flow.

to the target point. This routing path denoted as $\mathcal{P}\{P_s, B_{l_s}, T_{l_t}\}$ includes two parts.

1) The wire connecting P_s to the bend point B_{l_s}.
2) The vias to change routing metal layers from l_s to l_t and the wire connecting to T_{l_t}.

As shown in Fig. 8, l_s is 4 and l_t 1 in the path noted by green while l_s is 1 and l_t is 2 in the other sample path. Based on the above two parts of the 3-D L-shape pattern routing solution, we calculate the cost of $\mathcal{P}\{P_s, B_{l_s}, T_{l_t}\}$ with

$$c\big(\mathcal{P}\{P_s, B_{l_s}, T_{l_t}\}\big) = c_{\text{w}}(P_s, B, l_s) + c_{\text{v}}(B, l_s, l_t) + c_{\text{w}}\big(B, T_{l_t}, l_t\big). \tag{1}$$

Besides the path cost described above, we also consider the vias connecting parent two-pin net with the children two-pin nets, which are defined by the intranet ordering strategy described in Section II-D. Supposed that $P_s \rightarrow P_t$ has c children two-pin nets $P_s^{(i)} \rightarrow P_s, 1 \le i \le c$, we calculate *the bottom children cost* in (2) and this cost is not related to the target layer of T_i

$$c_{\text{bc}}(P_s, l_s) = \min_{0<l_1,\ldots,l_c\le L}\left\{c_{\text{v}}(P_s, l_s, l_1, \ldots, l_c) + \sum_{1\le i\le c} c^*(P_s^{(i)}, P_s, l_i)\right\} \tag{2}$$

where l_s is the source layer of P_s in the two-pin net $P_s \rightarrow P_t$ and $l_1, \ldots, l_c$ are all the layers of the children nets at the 2-D position as P_s; $c^*(P_s^{(i)}, P_s, l_i)$ means the minimum cost of the ith child two-pin net $P_s^{(i)} \rightarrow P_s$ on the l_i layer.

Having the bottom children costs and the path costs, we can compute the final $c^*(P_s, P_t, l_t)$ using the *dynamic programming procedure* described in the following:

$$c^*(P_s, P_t, l_t) = \min_{0<l_s\le L}\big\{c\big(\mathcal{P}\{P_s, B_{l_s}, T_{l_t}\}\big) + c_{\text{bc}}(P_s, l_s)\big\} \tag{3}$$

where l_s is the source layer of B_{l_s} and l_t is the target layer of T_{l_t} in this two-pin net.

The minimum cost of the whole ordered multipin net with the root edge $P_s^r \rightarrow P_t^r$ is defined as follows:

$$c^*\big(P_t^r\big) = \min_{0<l_r\le L} c^*\big(P_s^r, P_t^r, l_r\big) \tag{4}$$

where P_t^r means the root point and $P_s^r \rightarrow P_t^r$ is the root edge. We define the root point in Section II-D and illustrate the sample root point P_6 and sample root edge $P_5 \rightarrow P_6$ in Fig. 4.

We propose a GPU-friendly L-shape pattern routing algorithm for each two-pin net based on the above calculation flow. Our algorithm can compute all the L outputs $c^*(P_s, P_t, l_t)$ simultaneously using computation flow. Within each computation flow to get $c^*(P_s, P_t, l_t)$, our algorithm can enumerate all L candidate l_s at the same time. They can utilize the homogeneous GPUs threading resources well to enumerate $L \times L$ combinations for l_s and l_t simultaneously.

The weights of the edges $P_s \rightarrow B_{l_s}$ in the computation graph considers the *bottom children cost* and the edge cost connecting P_s to B. The formal formulation of l_sth entry of the edge weights vector $\vec{w}^{(1)}$ is as follows:

$$w_{l_s}^{(1)} = c_{\text{bc}}(P_s, l_s) + c_{\text{w}}(P_s, B, l_s), \ 0 < l_s \le L. \tag{5}$$

The $B_{l_s} \rightarrow T_{l_t}$ works as enumerating the combinations of the metal layer of B and the layer of T for all the candidate 3-D L-shape pattern routing paths. The entry of the edge weights matrix $\mathbf{W}^{(2)}$ at the l_sth row and the l_tth column is

$$w_{l_s,l_t}^{(2)} = c_{\text{v}}(B, l_s, l_t) + c_{\text{w}}\big(B, T_{l_t}, l_t\big), \ 0 < l_s \le L, \ 0 < l_t \le L \tag{6}$$

Following the 3-D L-shape pattern routing algorithm procedure, we can calculate $c^*(P_s, P_t, l_t)$ using the vector addition and minimum operations, which is much more friendly to GPU implementation:

$$c^*(P_s, P_t, l_t) = \min_{0<l_s\le L}\left\{w_{l_s}^{(1)} + w_{l_s,l_t}^{(2)}\right\}. \tag{7}$$

Furthermore, all the L minimum costs $c^*(P_s, P_t, l_t)$ with different l_t can be computed using (7) at the same time since there is no dependency among all the L calculation flows.

E. Pattern Routing Stage: GPU-Friendly 3-D Z-Shape Pattern Routing

As shown in Fig. 2, there are two common patterns in pattern routing approaches, where the L-shape pattern provides two candidate routing paths in 2-D space and $L \times L$ candidate routing paths in 3-D space, where L is the number of metal layers, since L-pattern has one single bend point and there are preferred routing directions in 3-D routing space. At the same time, there are two bend points in Z-shape patterns named the source bend point, and the target bend point since one connects to the source pin, and the other connects to the target pin. Note that once the position of the target bend point is determined, the location of the source bend point is determined accordingly. Therefore, Z-pattern can get $M+N-2$ candidate paths in 2-D space and $(M+N-2) \times (L \times L \times L)$ candidate routing paths on account of the two bend points, where M represents the width of the bounding box of the net on G and N is the height. The Z-shape pattern is like an intermediate state between the L-shape pattern and maze routing paths regarding the number of candidate paths. Table II defines the additional notations used in our GPU-friendly 3-D Z-shape pattern routing algorithms for the two-pin net $P_s \rightarrow P_t$.

The 3-D Z-shape pattern routing can provide more candidate routing paths than 3-D L-shape pattern routing, especially for the nets with a large bounding box. Following the formulation of GPU-friendly 3-D L-shape pattern routing, we define a

TABLE II
ADDITIONAL NOTATIONS FOR GPU-FRIENDLY Z-SHAPE PATTERN ROUTING

Notation	Description
$(B^{s(i)}, B^{t(i)})$	The i^{th} bend point pair of 2D Z-shape patterns.
$B_l^{s(i)}$ $B_l^{t(i)}$	The bend points on l^{th} layer of 3D patterns.
$c^{*(i)}(P_s, P_t, l_t)$	The minimum cost with $(B^{s(i)}, B^{t(i)})$.
$c^*(P_s, P_t, l_t)$	The minimum cost of $\mathcal{T}(P_s)$ attached with $P_s \rightarrow T_{l_t}$.

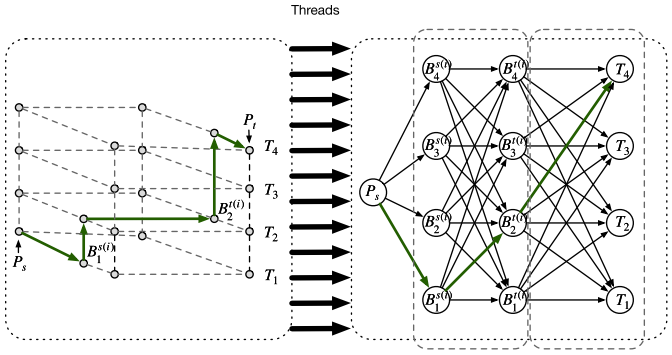


Fig. 9. GPU-friendly 3-D Z-shape pattern routing flow for ith candidate bend point pair $(B^{s(i)}, B^{t(i)})$; each 3-D Z-shape pattern routing solution is denoted as $\mathcal{P}\{P_s, B_{l_s}^{s(i)}, B_{l_t}^{t(i)}, T_{l_t}\}$. The sample routing path is colored by green.

GPU-friendly 3-D Z-shape pattern routing algorithm as shown in Fig. 9. The left part in Fig. 9 illustrates one of the solutions of $P_s \rightarrow P_t$ with 3-D Z-shape pattern routing algorithm and the candidate bend point pair $(B^{s(i)}, B^{t(i)})$, where i means the index of the candidate bend point pair and $1 \le i \le M+N-2$.

Besides l_s and l_t defined in the 3-D L-shape pattern routing algorithm, we use l_b to represent the layer of the wire connecting the source bend point and the target bend point in the 3-D Z-shape patterns. The 3-D Z-shape routing path includes three parts.

1) The wire connecting the source point P_s to the source bend point $B_{l_s}^{s(i)}$.
2) The vias to change routing metal layers from l_s to l_b and the wire connecting to the target bend point $B_{l_b}^{t(i)}$.
3) The vias to change routing metal layers from l_b to l_t and the wire connecting to the target point T_{l_t}.

In this sample path, l_s is 1, l_b is 2, and l_t is 4. As the same to the analysis in the 3-D L-shape pattern routing algorithm, we denote this candidate Z-shape pattern routing path as $\mathcal{P}\{P_s, B_{l_s}^{s(i)}, B_{l_b}^{t(i)}, T_{l_t}\}$. According to the above three parts, the formal formulation to calculate the cost of this path is as follows:

$$\begin{aligned} c\left(\mathcal{P}\left\{P_s, B_{l_s}^{s(i)}, B_{l_b}^{t(i)}, T_{l_t}\right\}\right) &= c_{\text{w}}\left(P_s, B^{s(i)}, l_s\right) \\ &+ c_{\text{v}}\left(B^{s(i)}, l_s, l_b\right) + c_{\text{w}}\left(B^{s(i)}, B^{t(i)}, l_b\right) \\ &+ c_{\text{v}}\left(B^{t(i)}, l_b, l_t\right) + c_{\text{w}}\left(B^{t(i)}, T_{l_t}, l_t\right). \end{aligned} \tag{8}$$

For each pair of bend points $(B^{s(i)}, B^{t(i)})$ in Z-shape patterns, we will generate the candidate flow i for this bend point pair. $c^{*(i)}(P_s, P_t, l_t)$ represents the minimum cost result of the two-pin net $P_s \rightarrow P_t$ in the ith candidate flow with the 3-D Z-shape pattern routing algorithm. Similar to (7), the calculation of $c^{*(i)}(P_s, P_t, l_t)$ is shown in the following:

$$\begin{aligned} &c^{*(i)}(P_s, P_t, l_t) \\ &= \min_{0<l_s,l_b\le L}\left\{c_{\text{bc}}(P_s, l_s) + c\left(\mathcal{P}\left\{P_s, B_{l_s}^{s(i)}, B_{l_b}^{t(i)}, T_{l_t}\right\}\right)\right\}. \end{aligned} \tag{9}$$

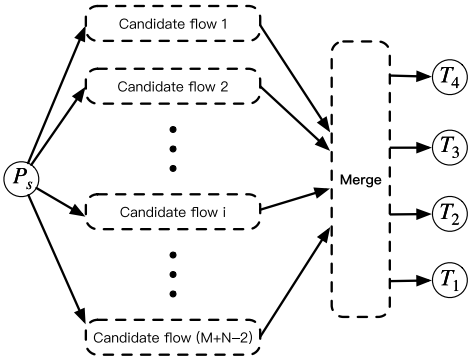


Fig. 10. Overall GPU-friendly 3-D Z-shape pattern routing flow.

As shown in Fig. 10, we propose a merge step to merge the results of all $M+N-2$ candidate flows. With the merge step, we can get the final minimum cost as follows:

$$c^*(P_s, P_t, l_t) = \min_{1\le i\le M+N-2} c^{*(i)}(P_s, P_t, l_t). \tag{10}$$

To better utilize the GPU resources, we also reformulate the computation in Z-shape patterns to the computation graph flow using the vector/matrix addition and minimum operation. Our proposed GPU-friendly Z-shape pattern routing algorithm for the ith candidate bend point pair $(B^{s(i)}, B^{t(i)})$ is shown in the right part of Fig. 9.

The weight of the edge $P_s \rightarrow B_{l_s}^{s(i)}$ includes the bottom children cost $c_{\text{bc}}(P_s, l_s)$ and the wire cost to connect P_s and $B^{s(i)}$. The formal formulation of l_sth of the edge weights vector $\vec{w}^{(1)}$ is as follows:

$$w_{l_s}^{(1)} = c_{\text{bc}}(P_s, l_s) + c_{\text{w}}\left(P_s, B^{s(i)}, l_s\right), \ 0 < l_s \le L. \tag{11}$$

The metal layer switch procedure at the source bend point is represented by the connection between $B_{l_s}^{s(i)}$ and $B_{l_b}^{t(i)}$. Based on the connection, we formulate the entry of the edge weights matrix $\mathbf{W}^{(2)}$ at the l_sth row and the l_bth column as follows:

$$w_{l_s,l_b}^{(2)} = c_{\text{v}}\left(B^{s(i)}, l_s, l_b\right) + c_{\text{w}}\left(B^{s(i)}, B^{t(i)}, l_b\right), \ 0 < l_s, l_b \le L. \tag{12}$$

Similar to the metal layer switch procedure at the source bend point, we can also define the metal layer switch procedure at the target bend point as the connection between $B_{l_b}^{t(i)}$ and T_{l_t}. The entry at l_bth row and the l_tth column of the edge weights matrix $\mathbf{W}^{(3)}$ is defined as follows:

$$w_{l_b,l_t}^{(3)} = c_{\text{v}}\left(B^{t(i)}, l_b, l_t\right) + c_{\text{w}}\left(B^{t(i)}, T_{l_t}, l_t\right), \ 0 < l_b, l_t \le L. \tag{13}$$

The minimum cost $c^{*(i)}(P_s, P_t, l_t)$ of the candidate bend point pair $(B^{s(i)}, B^{t(i)})$ can be calculated as (14) referring to (9)

$$c^{*(i)}(P_s, P_t, l_t) = \min_{0<l_s,l_b\le L}\left\{w_{l_s}^{(1)} + w_{l_s,l_b}^{(2)} + w_{l_b,l_t}^{(3)}\right\}. \tag{14}$$

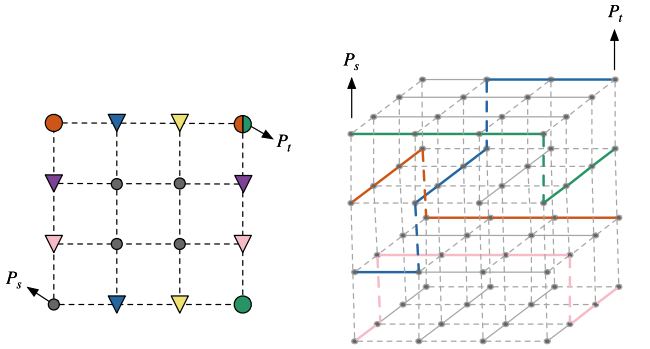


Fig. 11. Hybrid shape pattern routing; The left part shows all the bend point pairs and the right part contains four candidate 3-D solutions.

Having all the minimum cost of $M + N - 2$ candidate bend point pairs, we can finally get the $c^*(P_s, P_t, l_t)$ using the merge step in (10) which can also be computed as the vector minimum operation on GPU.

F. Pattern Routing Stage: GPU-Friendly Hybrid-Shape Pattern Routing

We combine the L-shape and Z-shape pattern routing to form a hybrid-shape pattern routing algorithm since the candidate paths in L-shape patterns are sometimes crucial for the routing path selection. The proposed hybrid-shape pattern routing enables the two-stage global router framework to obtain a better global routing solution with a little runtime overhead.

As illustrated in Fig. 2, the difference between the L-shape pattern and the Z-shape pattern is the number of bend points, where the L-pattern only has one bend point, while the Z-pattern gets two bend points.

We regard the L-shape pattern as a special case of the routing patterns with two bend points. We analyze this conclusion in 2-D space, and it is clear that the situation is similar in 3-D space. For all the $(M + N) - 2$ Z-shape candidate patterns, we only need to allocate the position of the target bend point along the two bounding box edges connected to the target point. One edge includes $M - 1$ candidate positions and the other gets $N - 1$ candidates. The position of the source bend point is determined automatically according to the position of the target bend point. We set the target point overlapping with the target bend point so that the source bend point gets two possible positions, resulting in two L-shaped shape patterns.

The left part of Fig. 11 illustrates the candidate bend point positions of our hybrid shape pattern routing algorithm. The colored triangle nodes represent a set of bend point pairs in Z-shape patterns, and the colors represent the corresponding relationships for pairs. The colored round nodes represent the two L-shape patterns, and the round node on P_t's position represents the target bend points overlapped with the target point. The right part of Fig. 11 is four candidate solutions for our 3-D hybrid-shape pattern routing algorithm. Note that, the color of paths on the right corresponds to the color of bend point pairs on the left. In this way, we can unify the definition of the hybrid shape pattern routing algorithm as the GPU-friendly Z-shape pattern routing algorithm designed in Section III-E with $M + N$ candidates bend point pairs.

TABLE III
ICCAD2019 BENCHMARKS

Bench	# nets	# G-cells
18test5	72394	619×613
18test8	179863	905×883
18test10	182000	606×522
19test7	358720	1053×1011
19test8	537577	1202×1138
19test9	895252	1337×1433

G. Parallel Rip-Up and Reroute Iterations

For several multipin nets, the pattern routing stage is never able to obtain a violation-free routing solution. To decrease the total amount of violations, multiple rip-up and reroute iterations will be performed. We simply need to rewire the nets with violations in our rip-up and reroute process after the pattern routing stage. We apply our task graph scheduler to utilize the parallelism among all these multipin nets for the runtime decreasing. Each multipin net is treated as a separate routing task, as distinct to the pattern routing stage, since the number of multipin nets in the pattern routing stage is much more than the number in the rip-up and reroute iterations. Then, we apply our task graph scheduler to these routing tasks based on the conflict relationship. Finally, all these routing tasks follow the execution order determined in the ordered task graph.

As a result, utilizing Taskflow [30] and the ordered task graph built by our proposed task graph scheduler, we can quickly maximize the parallelism of our rip-up and reroute iterations. Taskflow is a C++ tasking toolkit that uses the task dependency graph to automatically execute all the tasks. It utilizes the ordered task graph as the task dependency graph and execute all the tasks in maximum parallelism by utilizing CPU threading resources.

IV. Experimental Results

A. Experimental Setup

The framework was developed in C++/CUDA based on the open-source global router CUGR [3]. We conducted the experiments on a 64-bit Linux machine with Intel Xeon Gold 6226R CPU @ 2.90 GHz and 1 NVIDIA GeForce RTX 3090 GPU. ICCAD2019 benchmarks [26] were adopted to evaluate the performance.

To estimate the effectiveness of our proposed scheduler, we implemented our proposed task graph scheduler in both the pattern routing stage and the rip-up and reroute iterations. Meanwhile, we integrated our proposed two types of GPU-friendly pattern routing algorithms into the pattern routing stage separately to illustrate the strength of our methods.

B. Benchmark

The details for the ICCAD2019 benchmarks are listed in Table III. We only list half of the benchmarks since the other half, which end with "m," have the same number of nets and

TABLE IV
SORTING SCHEMES

ID	Sorting Scheme
0	Descending guide area size
1	Ascending guide area size
2	Descending bounding box half perimeter
3	Ascending bounding box half perimeter
4	Descending #pins
5	Ascending #pins

TABLE V
EXPERIMENTAL RESULTS OF DIFFERENT SORTING SCHEMES; THE SORTING SCHEMES ARE ONLY SUBSTITUTED IN THE RIP-UP AND REROUTE ITERATIONS. THE BEST IS MARKED AS BOLD AND THE SECOND BEST IS NOTED AS BLUE

Bench	Tech	TOTAL (s)	PATTERN (s)	MAZE (s)	Score ($\times 10^7$)
18test10	0	150.842	8.642	107.311	4.28
	1	143.102	8.63	100.501	4.28
	2	**119.412**	8.664	**75.815**	4.28
	3	120.084	8.655	76.988	4.28
	4	137.557	8.651	93.874	4.28
	5	131.658	8.883	86.723	4.28
18test10m	0	197.707	4.892	136.109	**4.45**
	1	187.584	4.866	127.72	4.48
	2	**173.811**	4.674	**114.838**	4.48
	3	183.41	4.954	124.884	4.47
	4	192.467	4.902	129.837	4.49
	5	186.261	4.898	127.028	4.48

number of G-cells, and the only difference between them is the number of metal layers. This set of benchmarks includes the small design with 70k nets to the extensive design with nearly 900k nets. The scalability of the ICCAD2019 benchmark helps us better evaluate our approaches.

C. Sorting Scheme

The experimental results with various sorting schemes, listed in Table IV, reveal that net ordering influences the final solution quality performance and the running time. We select six different schemes that are solely applied in the rip-up and reroute iterations to highlight the influence of net ordering while maintaining the pattern routing stage process. Table V displays the experimental results. The running time of the rip-up and reroute iterations varies depending on the sorting scheme since the routing order influences the routing process and an earlier routed net may hamper the continue net with fewer routing resources.

Furthermore, we use a weighted sum score that takes three metrics into account: 1) wirelength; 2) the number of vias; and 3) the number of shorts violations, to indicate the solution quality of a global routing engine. The formal formulation to compute the score s is as follows:

$$s = \alpha W + \beta V + \gamma S \tag{15}$$

where W means the wirelength, V for the number of vias, and S as the number of shorts violations. Additionally, α, β, and γ stand for the three weights of wirelength, vias number, and shorts violation number. (In our experiments, we set α to 0.5, β to 4, and γ to 500, considering the order of magnitude of different metrics.)

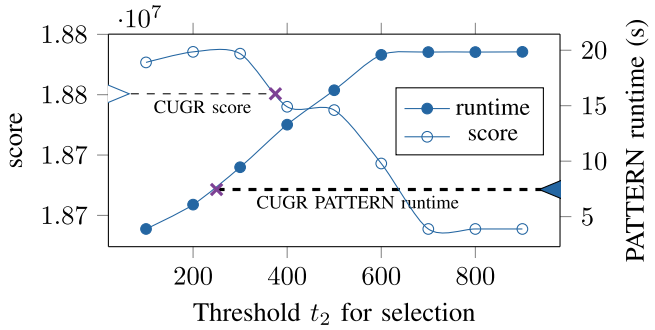


Fig. 12. Performance with different t_2 for the design 18test5m, where t_1 is 100; the solid circle line represents PATTERN runtime while the other is the global routing score. The *thick dashed line* is the baseline PATTERN runtime of CUGR and the other is the baseline score.

To be more thorough, we choose two benchmarks to analyze the influence of routing order, `18test10` with nine metal layers and `18test10m` with only five metal layers. TOTAL is used in Table V to indicate total running time; PATTERN is used to represent the running time of the pattern routing stage, and MAZE is used to describe the running time of rip-up and reroute iterations. According to the experimental results in Table V, adopting bounding box half perimeter as the measurement to sort nets can improve the running time for both of these two benchmarks. Besides the running time, solution quality matters in the global routing stage. Overall, the ascending bounding box half perimeter is a better sorting scheme for the global router.

D. Selection

We observed that when the hybrid shape pattern routing algorithm is applied to all the two-pin nets, it suffered the performance of our global router in both acceleration performance and the final solution quality performance. The acceleration performance is suffered from a few tremendous nets, which are only less than 0.01% of all the nets. Still, they can generate thousands of candidate bend point pairs in the hybrid shape pattern routing algorithm. As for the solution quality performance, when we apply the hybrid-shape pattern routing algorithm to all the two-pin nets regardless of their size, the small nets executed first will impede the routing for the following more giant nets since the routing resources are limited. Based on this observation and analysis, we develop a selection technique in our proposed hybrid-shape pattern routing algorithm to improve the performance of our global router. First, we set two thresholds, t_1 and t_2, to split all the two-pin nets into three parts, small nets, medium nets, and large nets, according to the bounding box size of them. Noted that, we use half perimeter wirelength (HPWL) to represent the size of the bounding box since the number of candidates flows in the hybrid-shape pattern routing algorithm is related to HPWL. To better explain the choice of the split thresholds, we show the variation trend of 18test5m's PATTERN runtime and global routing score in Fig. 12 with the fixed t_1, 100, and changing t_2 from 100 to 1000. It is reasonable that with a larger t_2, the performance is better but the running time is longer since the hybrid-shape pattern routing can consider

TABLE VI
ABLATION STUDY FOR FASTGR^{H}

Bench	TOTAL runtime(s)		PATTERN runtime(s)		# Nets to rip up		# Shorts		
	w/o selection	selection	w/o selection	selection	w/o selection	selection	w/o selection	selection	Improved (%)
18test5	61.614	45.240	37.513	27.774	316	300	0	0	
18test5m	80.415	40.509	16.678	13.084	7215	8766	3167.5	3135	1.026
18test8	242.495	133.717	105.56	53.110	1780	1937	0	0	
18test8m	227.588	141.511	30.059	23.690	26095	30498	5309	5360	-0.961
18test10	297.022	144.570	58.915	38.503	5225	5538	0	0	
18test10m	357.176	169.670	19.374	15.858	29624	35297	1214	1000	17.628
19test7	559.226	231.654	385.36	85.943	1925	3803	0	0	
19test7m	320.682	210.831	51.303	45.032	17220	18438	4260	4028.5	5.434
19test8	557.069	240.520	399.71	93.800	804	1823	0	0	
19test8m	554.969	317.376	86.66	43.210	21068	27302	6959.5	6822	1.976
19test9	746.643	320.815	525.21	103.690	1052	1699	98.5	23.5	76.142
19test9m	696.071	498.958	106.9	52.054	29016	37792	3765.5	3692	1.952
Average	1.888	1.000	2.304	1.000	0.789	1.000			14.742

TABLE VII
OVERALL RESULTS ON ICCAD 2019 BENCHMARKS

Bench	Total runtime (s)					Score		
	CUGR	FastGR^{L} [15]	Speedup	FastGR^{H}	Speedup	CUGR	FastGR^{L} [15]	FastGR^{H}
18test5	80.645	24.858	3.244×	45.240	1.783×	16931900	16919500	16880800
18test5m	83.49	33.457	2.495×	40.509	2.061×	18760300	18774900	18705100
18test8	260.982	97.568	2.675×	133.717	1.952×	40880300	40881100	40825200
18test8m	250.26	131.395	1.905×	141.511	1.768×	42924700	42897000	42728700
18test10	354.347	110.811	3.198×	144.570	2.451×	42628400	42592100	42575900
18test10m	339.471	165.682	2.049×	169.670	2.001×	44193400	44579100	44275300
19test7	527.955	181.445	2.910×	231.654	2.279×	71852400	71887200	71763600
19test7m	340.489	179.115	1.901×	210.831	1.615×	67958400	67923300	67639200
19test8	529.193	173.299	3.054×	240.520	2.200×	113944000	113929000	113844000
19test8m	520.547	320.228	1.626×	317.376	1.640×	114748000	114362000	114545000
19test9	842.18	250.466	3.362×	320.815	2.625×	175448000	175523000	175367000
19test9m	632.577	434.852	1.455×	498.958	1.268×	173154000	173339000	173002000
Average			2.489×		1.970×	1.000	1.000	0.998

more candidate routing paths than the L-shape one. As shown in Fig. 12, when t_2 is smaller than 250, hybrid-shape pattern routing can achieve runtime improvement compared to CUGR. Furthermore, the solution quality will be improved when t_2 is larger than 380. Therefore, we finally choose 100 and 500 as the split thresholds in FastGR^{H}.

According to the split results, the small nets account for around 99%, the medium nets account for around 1%, and the large nets only get nearly 0.1%. After that, we only apply our proposed GPU-friendly hybrid-shape pattern routing algorithm in the medium nets, and use our proposed GPU-friendly L-shape pattern routing algorithm to the rest nets.

According to the experimental results, the selection technique improves both the acceleration and quality performance of our proposed hybrid-shape pattern routing algorithm. Table VI plots the running time and quality comparison between FastGR^{L} and FastGR^{H} without the selection technique. We can get 2.304× acceleration by applying selection to the pattern routing stage, while the number of nets with violations that should be passed to rip up and reroute iteration increases by 21.1%. Therefore, for the total running time, we can only achieve 1.888× speedup compared with FastGR^{H} without selection. As for the quality performance, we can get a 14.742% improvement on average concerning the number of shorts, which is a significant metric to reflect the routability.

E. Acceleration

We conduct experiments on 12 distinct benchmarks from the ICCAD2019 contest with advanced nodes. And, we find out that half of them (end with "m") contain only five metal layers, while the other six contain nine. Furthermore, as discussed in Section IV-C, we eventually arrange all the routing tasks in both stages with ascending bounding box half perimeter to obtain better running time and solution quality performance.

In order to show the acceleration performance of our task graph scheduler and our two GPU-friendly pattern routing algorithms, we assess total runtime, pattern routing runtime, and the runtime of the rip-up and reroute iterations. In addition, we display the solution quality using the score provided in (15). The FastGR with our proposed GPU-friendly L-shape pattern routing algorithm is named FastGR^{L} [15] and FastGR^{H} is short for our global router integrated with our GPU-friendly hybrid-shape pattern routing algorithm.

We evaluate the total runtime, pattern routing runtime, and the runtime of the rip-up and reroute iterations to demonstrate the acceleration performance of our proposed task graph scheduler and our developed two GPU-friendly pattern routing algorithms. Also, we evaluate the solution quality using the score defined in (15).

Table VII plots the overall performance with total running time and the solution quality on ICCAD2019 benchmarks. By

TABLE VIII
BREAKDOWN RUNTIME RESULTS ON ICCAD 2019 BENCHMARKS

Bench	PATTERN runtime (s)					# Nets to rip up			MAZE runtime (s)				
	CUGR	FastGR^{L}	Speedup	FastGR^{H}	Speedup	CUGR	FastGR^{L}	FastGR^{H}	CUGR	FastGR^{L}	Speedup	FastGR^{H}	Speedup
18test5	45.111	4.967	9.081×	27.774	1.624×	868	783	300	21.588	5.219	4.136×	2.320	9.304×
18test5m	7.445	3.314	2.247×	13.084	0.569×	11388	11346	8766	63.32	16.117	3.929×	13.604	4.654×
18test8	118.117	8.983	13.149×	53.110	2.224×	2847	2904	1937	108.986	52.692	2.068×	46.397	2.349×
18test8m	18.292	5.715	3.201×	23.690	0.772×	34260	34184	30498	201.58	79.987	2.520×	77.740	2.593×
18test10	124.533	10.410	11.963×	38.503	3.234×	5825	5856	5538	199.777	70.222	2.845×	73.873	2.704×
18test10m	18.485	6.788	2.723×	15.858	1.166×	38813	38407	35297	291.58	106.097	2.748×	103.631	2.814×
19test7	223.947	15.556	14.396×	85.943	2.606×	5334	5044	3803	241.787	98.780	2.448×	77.569	3.117×
19test7m	34.208	8.658	3.951×	45.032	0.760×	22353	22069	18438	244.686	105.852	2.312×	104.355	2.345×
19test8	333.965	17.612	18.962×	93.800	3.560×	2734	2527	1823	96.888	53.312	1.817×	42.909	2.258×
19test8m	51.922	11.982	4.333×	43.210	1.202×	31326	30590	27302	371.748	202.516	1.836×	169.857	2.189×
19test9	561.337	24.481	22.929×	103.690	5.414×	2420	2365	1699	129.707	74.764	1.735×	60.311	2.151×
19test9m	88.971	17.976	4.949×	52.054	1.709×	42957	42509	37792	400.227	247.518	1.617×	283.915	1.410×
Average			9.324×		2.070×	1.000	0.976	0.767			2.501×		3.157×

utilizing our proposed task graph scheduler and the GPU-friendly L-shape pattern routing algorithm, we can get an overall 2.489× acceleration over the widely used modern two-stage global router [3]. By conducting the GPU-friendly hybrid-shape pattern routing algorithm and the scheduler, we can still obtain 1.970× speed up compared with the baseline.

To better illustrate the impact on the different stages, Table VIII lists the pattern routing running time in the pattern routing stage and the maze routing running time of three rip-up and reroute iterations named PATTERN runtime and MAZE runtime, respectively. As for the GPU-friendly pattern routing algorithms, we apply the zero-copy technique [31] of the CUDA library in our implementation to shorten the data transmission running time between the CPU and the GPU within 1 s. Therefore, the PATTERN runtime in Table VIII primarily reflects the efficiency of our proposed GPU-friendly pattern routing algorithms.

With the zero-copy technique, Table VIII shows that our proposed GPU-friendly L-shape pattern routing algorithm can bring 9.324× speedup. Meanwhile, the proposed GPU-friendly hybrid-shape pattern routing algorithm can achieve 2.070× acceleration on average compared with the sequentially executed strategy. The reduction of the speedup is because the hybrid-shape pattern routing algorithm considers $\sum_{e\in E}((M_e+N_e)\times L\times L\times L\times L)$ candidate routing paths compared with both CUGR [3] and FastGR^{L} [15], which contain $\sum_{e\in E}(L\times L)$ candidate routing paths in the L-shape pattern routing algorithm. We set M_e, N_e to represent the width and the height of net e on the global routing grid graph G and L is the number of metal layers. Further analysis of the pattern routing running time reveals that the performance of our GPU-friendly pattern routing algorithms may be determined by design scale. If the case has a larger design scale, the performance will be better. In addition, modern circuits' metal layers exceed five thus our algorithms are desirable in the industry.

Respecting the rip-up and reroute iterations, we count the number of nets after the pattern routing stage, which is related to the MAZE runtime since only the nets with violations will be passed into rip-up and reroute iterations. In comparison with the widely adopted batch-based parallelization strategy, our GPU-friendly L-shape pattern routing algorithm can reduce the number of nets with violation by 2.4%. On the other hand, the GPU-friendly hybrid-shape pattern routing strategy can significantly reduce the number of nets with violations by 23.3% on average. Both of them illustrate that our proposed pattern routing framework can improve the solution quality of the pattern routing stage.

In the case with a similar number of nets to rip up, the task scheduler can contribute to 2.501× speedup over the widely adopted batch-based parallelization strategy on the CPU. By applying the hybrid-shape pattern routing algorithm in the pattern routing stage, our proposed framework can largely reduce the number of nets to rip up, and the running time of maze routing iterations can achieve a 3.157% on average compared to the baseline. This improvement between FastGR^{L} [15] and FastGR^{H} mainly gains from the reduction of the number of nets to rip up.

F. Time Complexity Analysis

The time complexity is analyzed as follows. The hybrid-shape pattern routing algorithm considers $\sum_{e\in E}((M_e+N_e)\times L\times L\times L)$ candidate routing paths compared with the L-shape pattern routing algorithm with $\sum_{e\in E}(L\times L)$ candidate routing paths, where E represents the set of all the two-pin nets. Considering executing all the multipin nets in sequential, the time complexity of our proposed hybrid-shape pattern routing algorithm gets an $\mathcal{O}(|E|(M+N)L^3)$ computational amount, where M and N are the width and height of the 2-D global routing grid graph, respectively. In that case, the L-shape pattern routing algorithm gets $\mathcal{O}(|E|L^2)$ computational amount.

Our developed computation graph flow methods enumerate all the layer combinations at the same time, which can help reduce the time complexity to $\mathcal{O}(|E|(M+N))$ and $\mathcal{O}(|E|)$, respectively, in an ideal situation with enough resources. Furthermore, with the help of the framework described in Fig. 10, we can enumerate all the $(M+N)$ candidates simultaneously when the computation resource is enough so that the time complexity of our proposed hybrid-shape pattern routing can be reduced to $\mathcal{O}(|E|)$.

On the other hand, the additional merge step for all $(M+N)$ candidates with this structure costs $\mathcal{O}(\log(M+N))$ by using divide-and-conquer. In our experiment, we implement the merge step by searching all results, and the cost of the merge

TABLE IX
SOLUTION QUALITY RESULTS

Bench	Score		Wirelength		# Vias		# Shorts		
	FastGR^{L}	FastGR^{H}	FastGR^{L}	FastGR^{H}	FastGR^{L}	FastGR^{H}	FastGR^{L}	FastGR^{H}	Improved (%)
18test5	16919500	**16880800**	26988200	**26915900**	856362	**855723**	0	0	
18test5m	18774900	**18705100**	27942700	**27799600**	**802385**	809459	3188	**3135**	1.662
18test8	40881100	**40825200**	64359900	**64211700**	**2174240**	2179840	8.5	**0**	100.000
18test8m	42897000	**42728700**	64554600	**64441200**	**1948740**	1957020	5649.5	**5360**	5.124
18test10	42592100	**42575900**	66718200	**66677100**	**2308250**	2309320	0	0	
18test10m	44579100	**44275300**	71071700	**71014000**	**2059820**	2067080	1608	**1000**	37.811
19test7	71887200	**71763600**	118744000	**118495000**	**3128750**	3129020	0	0	
19test7m	67923300	**67639200**	107115000	**106577000**	**3080660**	3084060	4086.5	**4028.5**	1.419
19test8	113929000	**113844000**	181854000	**181687000**	5750370	**5750260**	0	0	
19test8m	**114362000**	114545000	177638000	**177284000**	**5612590**	5623080	**6185.5**	6822	-10.290
19test9	175523000	**175367000**	274106000	**273884000**	9603400	**9603200**	112.5	**23.5**	79.111
19test9m	173339000	**173002000**	267786000	**267304000**	**9359980**	9375940	4013	**3692**	7.999
Average									27.855

TABLE X
QUALITY COMPARISON AFTER DETAILED ROUTING. THE BEST IS MARKED AS BOLD AND THE SECOND BEST IS NOTED AS BLUE

Bench	Wirelength			# Vias			# Shorts			# Spacing Vios		
	CUGR	FastGR^{L}	FastGR^{H}	CUGR	FastGR^{L}	FastGR^{H}	CUGR	FastGR^{L}	FastGR^{H}	CUGR	FastGR^{L}	FastGR^{H}
18test5	28409900	28397700	**28325500**	927246	927434	**926470**	**386.5**	387	389	**76**	96	109
18test5m	28796200	28822800	**28776600**	**915100**	916493	916747	378	396.5	**377**	15	27	**14**
18test8	67685300	67660100	**67575900**	2345970	**2345440**	2347850	420.1	409.625	**406.6**	**48**	60	88
18test8m	66546100	66335000	**66304800**	2246450	**2242980**	2244520	412.25	418.75	**412**	69	88	**56**
18test10	70461700	70448600	**70433400**	**2496290**	2497280	2496680	**300.85**	526.375	380.425	**982**	1383	1153
18test10m	**72910700**	73435300	73401500	2434060	**2430150**	2430670	**8985.7**	12855.1	14085.5	**13606**	18790	19762
19test7	125584000	125664000	**125512000**	4045160	4047400	**4042230**	2203.35	**2102.88**	2134.85	7423	**6897**	7048
19test7m	**112763000**	112969000	112806000	4035500	4036730	**4032970**	2491.5	2529	**2462**	7680	**7392**	7439
19test8	192915000	192901000	**192764000**	6413290	6412920	**6411350**	**1212.5**	1224.95	1338.45	**4528**	4621	4704
19test8m	187050000	**186992000**	187127000	**6353690**	6362480	6362220	2324.5	3133	**1998.5**	5730	6354	**5632**
19test9	292672000	292710000	**292542000**	**10684300**	10687100	10684900	**2891.15**	2962.78	2912.55	**11725**	11918	11806
19test9m	**283354000**	283666000	283454000	**10579600**	10586900	10588800	4396.5	4155.5	**4096**	13828	13342	**12785**

step will be $\mathcal{O}(M+N)$. Finally, our GPU-accelerated hybrid-shape pattern routing algorithm can get $\mathcal{O}(|E|+M+N)$ time with enough computation resource, and the GPU-accelerated L-shape pattern routing algorithm costs $\mathcal{O}(|E|)$.

Based on the above analysis, our computation graph flow for routing only needs additional merge cost when extending more bend points in the modern routing algorithm with enough computational resources.

G. Evaluation on Hybrid Shape Patterns

The effectiveness of the proposed hybrid-shape pattern routing algorithm is verified by applying the hybrid shape pattern routing FastGR^{H} or the L-shape pattern routing FastGR^{L} [15] in our FastGR framework. Table IX shows the comparison of the detailed solution quality. It establishes that the global router with hybrid-shape pattern routing algorithm FastGR^{H} can beat the global router with L-shape pattern routing algorithm FastGR^{L} [15] with respect to the score defined in (15) and wirelength on most designs. Since our hybrid-shape pattern routing algorithm considers Z-shape patterns as the candidate routing paths, the number of vias increases reasonably. The most important metric for global routing solution quality is the number of shorts, representing the degree of violations. By replacing the L-shape pattern routing algorithm with our proposed hybrid-shape pattern routing algorithm, we can achieve a 27.855% improvement on average in the number of shorts. Note that especially for the designs with a few shorts, FastGR^{H} optimizes all the shorts, which is a significant improvement since a few shorts usually need to be manually fixed by the engineers, which takes a lot of time. In contrast, the large number of shorts still require more placement and routing (PnR) flow iterations and the more candidate routing paths only optimize a few rate of violations in that case.

H. Detailed Routing Performance

Since the global routing solution is only a guide for the detailed router, the final detailed routing solution quality may not be improved as the same as the global routing solution quality. To further evaluate the solution performance, Dr. Cu [4] is applied to conduct detailed routing under the guide of the global routing solution. The corresponding detailed routing results including wirelength, the number of vias, the number of shorts, and the number of spacing violations are listed in Table X. As for the wirelength, our FastGR framework outperforms CUGR on most designs. Furthermore, FastGR can obtain comparable detailed routing performance with CUGR in many aspects (including the number of vias, the number of shorts, and the number of spacing violations) as shown in Table X. On the other hand, compared with FastGR^{L}, FastGR^{H} can maintain a similar number of vias with better wirelength and routability performance.

V. Conclusion

In this article, we propose an efficient global routing framework, FastGR, accelerated for CPU–GPU platforms. We propose two GPU-friendly pattern routing algorithms and a heterogeneous task graph scheduler. The framework includes a fast version FastGR^{L}, which can obtain 2.489× acceleration with a 9.324× speedup over the sequential 3-D L-shape pattern routing algorithm on the CPU. We also develop a quality-oriented version FastGR^{H} to get a 27.855% improvement in respect of the number of shorts and maintain a 1.970× speedup at the same time. This article's results highlight the importance of GPU-accelerated kernel algorithms and the task scheduler for Internet ordering in routing. An adequate fuse of them can assist in reducing design cycles and improve the solution quality at the same time. In the future, we plan to extend the task graph scheduler to other critical stages and exploit the power of GPU acceleration in the VLSI flow.

References

[1] J. He, U. Agarwal, Y. Yang, R. Manohar, and K. Pingali, "SPRoute 2.0: A detailed-routability-driven deterministic parallel global router with soft capacity," in *Proc. ASPDAC*, 2022, pp. 586–591.

[2] A. B. Kahng, L. Wang, and B. Xu, "TritonRoute: The open-source detailed router," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 3, pp. 547–559, Mar. 2021.

[3] 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. DAC*, 2020, pp. 1–6.

[4] G. Chen, C.-W. Pui, H. Li, and E. F. 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.

[5] X. He et al., "Ripple 2.0: High quality routability-driven placement via global router integration," in *Proc. DAC*, 2013, pp. 1–6.

[6] J. Hu, J. A. Roy, and I. L. Markov, "Completing high-quality global routes," in *Proc. ISPD*, 2010, pp. 35–41.

[7] J. Soukup, "Global router," in *Proc. DAC*, 1979, pp. 481–484.

[8] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh, "Pattern routing: Use and theory for increasing predictability and avoiding coupling," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 7, pp. 777–790, Jul. 2002.

[9] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "FastRoute: An efficient and high-quality global router," in *Proc. VLSI Des.*, 2012, p. 14.

[10] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: Global router with efficient via minimization," in *Proc. ASPDAC*, 2009, pp. 576–581.

[11] M. D. Moffitt, "MaizeRouter: Engineering an effective global router," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 11, pp. 2017–2026, Nov. 2008.

[12] Y. Xu and C. Chu, "MGR: Multi-level global router," in *Proc. ICCAD*, 2011, pp. 250–255.

[13] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier, "All-pairs shortest path algorithms for planar graph for GPU-accelerated clusters," *J. Parallel Distrib. Comput.*, vol. 85, pp. 91–103, Nov. 2015.

[14] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 940–952, 2013.

[15] S. Liu et al., "FastGR: Global routing on CPU-GPU with heterogeneous task graph scheduler," in *Proc. DATE*, 2022, pp. 760–765.

[16] C. J. Alpert, T. C. Hu, J.-H. Huang, A. B. Kahng, and D. Karger, "Prim-Dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 7, pp. 890–896, Jul. 1995.

[17] 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.

[18] C. J. Alpert et al., "Prim-Dijkstra revisited: Achieving superior timing-driven routing trees," in *Proc. ISPD*, 2018, pp. 10–17.

[19] G. Chen and E. F. Y. Young, "SALT: Provably good routing topology by a novel steiner shallow-light tree algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 6, pp. 1217–1230, Jun. 2020.

[20] J.-R. Gao, P.-C. Wu, and T.-C. Wang, "A new global router for modern designs," in *Proc. ASPDAC*, 2008, pp. 232–237.

[21] Y. Han, D. M. Ancajas, K. Chakraborty, and S. Roy, "Exploring high-throughput computing paradigm for global routing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 1, pp. 155–167, Jan. 2014.

[22] M. M. Ozdal and M. D. F. Wong, "Archer: A history-based global routing algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 4, pp. 528–540, Apr. 2009.

[23] Y.-J. Chang, Y.-T. Lee, and T.-C. Wang, "NTHU-Route 2.0: A fast and stable global router," in *Proc. ICCAD*, 2008, pp. 338–343.

[24] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "A parallel integer programming approach to global routing," in *Proc. DAC*, 2010, pp. 194–199.

[25] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Scalable 3D global routing using integer programming," in *Proc. DAC*, 2009, pp. 320–325.

[26] S. Dolgov, A. Volkov, L. Wang, and B. Xu, "2019 CAD contest: LEF/DEF based global routing," in *Proc. ICCAD*, 2019, pp. 1–4.

[27] J. Hu and S. S. Sapatnekar, "A survey on multi-net global routing for integrated circuits," *Integration*, vol. 31, no. 1, pp. 1–49, 2001.

[28] C.-P. Hsu, "A new two-dimensional routing algorithm," in *Proc. DAC*, 1982, pp. 46–50.

[29] L. C. Abel, "On the ordering of connections for automatic wire routing," *IEEE Trans. Comput.*, vol. C-21, no. 11, pp. 1227–1233, Nov. 1972.

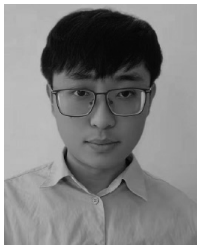
[30] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-TaskFlow: Fast task-based parallel programming using modern C++," in *Proc. IPDPS*, 2019, pp. 974–983.

[31] "Zero copy technique." NVDIA. Accessed: May 2009. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#zero-copy

Siting Liu received the B.S. degree from the Department of Computer Science, Huazhong University of Science and Technology, Wuhan, China, in 2020. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong.

She is a visiting student with the School of Integrated Circuits, Peking University, Beijing, China. Her current research interests include deep learning applications and GPU acceleration in physical design.

Ms. Liu received the Best Paper Award from DATE 2022 and the Best Paper Award Nomination from DATE 2021.

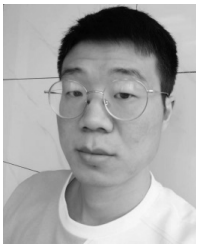


Yuan Pu received the B.S. degree in computer science from The Chinese University of Hong Kong, Hong Kong, in 2022.

He is currently a Research Assistant with the CSE Department, The Chinese University of Hong Kong, supervised by Prof. B. Yu. His research interest includes machine learning in EDA and hardware/algorithm co-optimization.

Peiyu Liao received the B.S. degree from the School of Mathematical Sciences, Zhejiang University, Hangzhou, China, in 2017, and the M.S. degree from the School of Engineering, The Hong Kong University of Science and Technology, Hong Kong, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

He is a visiting student with the School of Integrated Circuits, Peking University, Beijing, China. His current research interests include high-performance computing and numerical optimization in physical design.



Hongzhong Wu received the Ph.D. degree from the University of Science and Technology of China, Hefei, China.

He is currently a Senior Engineer with HiSilicon, Shenzhen, China.

Rui Zhang received the Ph.D. degree from the Institute of Semiconductor, Chinese Academy of Sciences, Beijing, China.

He is currently a Senior Engineer with the EDA Solutions Development Department, HiSilicon, Shenzhen, China. His research interests include design for manufacturing, floorplan automation, macro placement, and facilitating the use of artificial intelligent techniques in EDA solutions.

Zhitang Chen received the bachelor's degree from the Department of Automation, Sun Yat-sen University, Guangzhou, China, in 2010, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, in 2014.

He is a Researcher with the Noah's Ark Lab, Huawei Technologies, Hong Kong. His current research interests include kernel methods, deep learning, reinforcement learning, and multiagent systems and their applications to computer networks.

Wenlong Lv received the Ph.D. degree from Fudan University, Shanghai, China, in 2019.

He is currently a Principal Engineer with Huawei Technologies Company, Shenzhen, China. His main research interest is the combination of machine learning and EDA technologies.

Yibo Lin (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiaotong University, Shanghai, China, in 2013, and the Ph.D. degree from the Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX, USA, in 2018.

He is currently an Assistant Professor with the School of Integrated Circuits, Peking University, Beijing, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.

Bei Yu (Senior Member, IEEE) received the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 2014.

He is currently an Associate Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

Dr. Yu received nine Best Paper Awards from DATE 2022, ICCAD 2021 and 2013, ASPDAC 2021 and 2012, ICTAI 2019, Integration, the VLSI Journal in 2018, ISPD 2017, SPIE Advanced Lithography Conference 2016, and seven ICCAD/ISPD Contest Awards. He has served as the TPC Chair of ACM/IEEE Workshop on Machine Learning for CAD, and in many journal editorial boards and conference committees. He is an Editor of IEEE TCCPS NEWSLETTER.