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$^R$ 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$^Q$ 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.
while incurring a little runtime overhead. We develop two variants of our global routing framework: the runtime-oriented version FastGRL and the quality-oriented version FastGRH.

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 FastGRL 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 FastGRH reduces the number of shorts by 27.855% over the runtime-oriented version FastGRL [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.

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.
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, 19test7 is an MAZE-dominated design, and 19test9m 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 P6 as the random root in Fig. 4(a). Then, beginning with P6, we conduct DFS traversal. The DFS traversal accesses all the nodes in the following order: P6, P5, P4, P3, P2, and P1. As illustrated in Fig. 4(b), we mark all the two-pin nets in the reverse sequence e1, e2, e3, e4, and e5, 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

![Figure 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.](image)

![Figure 4: Example of intranet ordering. (a) Multiple two-pin nets. (b) Two-pin nets with order.](image)

![Figure 5: Example of intranet ordering. (a) Multiple two-pin nets. (b) Two-pin nets with order.](image)
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 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 conflicting 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 conflict 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

---

**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: $e \leftarrow \text{nets}[0]$;
2: Remove $e$ from nets and declare a new empty batch $\text{batch} \leftarrow \{e\}$;
3: for $e_i \in \text{nets}$ do
4:  \quad if $e_i$ has no conflict with all the nets in $\text{batch}$ then
5:  \quad  \quad Push $e_i$ into batch;
6:  \quad Remove $e_i$ from nets;
7:  \quad end if
8: end for
9: return batch;

---

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.
a 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 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], \( c_w(u, v, l) \) 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, \( c_v(u, l_1, l_2) \) 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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>The number of metal layers.</td>
</tr>
<tr>
<td>B</td>
<td>The bend point of 2D L-shape patterns</td>
</tr>
<tr>
<td>B_i</td>
<td>The bend points on the ith layer of 3D L-shape patterns.</td>
</tr>
<tr>
<td>T_i</td>
<td>The point on the ith layer with the same 2D position as P_i.</td>
</tr>
<tr>
<td>c(T(P))</td>
<td>The sub-tree rooted at P_i in the ordered multi-pin net.</td>
</tr>
<tr>
<td>c(·)</td>
<td>The cost of a routing path for a single two-pin net.</td>
</tr>
<tr>
<td>c_{bc}(P_i,l)</td>
<td>The bottom children cost with P_i in the lth layer.</td>
</tr>
<tr>
<td>c^*(P_s,P_t,l)</td>
<td>The minimum cost of Ω(P_s) attached with P_s → T_l.</td>
</tr>
<tr>
<td>w(i)</td>
<td>Weight vector of part i in our computation graph flow.</td>
</tr>
<tr>
<td>W(i)</td>
<td>Weight matrix of part i in our computation graph flow.</td>
</tr>
</tbody>
</table>

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) simultaneously using computation flow. Within each computation flow to get c^*(P_s,P_t,l), our algorithm can enumerate all L candidate l at the same time. They can utilize the homogeneous GPUs threading resources well to enumerate L × L combinations for l_i and l_t simultaneously.

The weights of the edges P_s → B_i in the computation graph considers the bottom children cost and the edge cost connecting P_s to B. The formal formulation of l_i entry of the edge weights vector w^{(1)} is as follows:

$$w^{(1)}_{l_i} = c_{bc}(P_s,l) + c_w(P_s,B_i)$$

The B_i → T_l 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 W(2) at the l_i row and the l_t column is

$$w^{(2)}_{l_i,l_t} = c_v(B,l_i,l_t) + c_w(B,T_l,l_t), 0 < l_i < L, 0 < l_t < L$$

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

$$c^*(P_s,P_t,l) = \min_{0 < l_i < L} \left\{ w^{(1)}_{l_i} + w^{(2)}_{l_i,l_t} \right\}$$

Furthermore, all the L minimum costs c^*(P_s,P_t,l) with different l_i 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 × 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 → 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
The flow of 3-D Z-shape pattern routing is illustrated in Fig. 9. The left part in Fig. 9 shows one of the solutions for each bend point pair \((P_s, B_{ls}^i, B_{lt}^j)\); each 3-D Z-shape pattern routing solution is denoted as \(P(P_s, B_{ls}^i, B_{lt}^j, T_{lt})\). The sample routing path is colored by green.

### Pattern Routing Algorithm

**Algorithm:**

1. **Input:** Source and target bend points \((P_s, P_t)\), bend points on \(i\)th layer \(B_{ls}^i, B_{lt}^j\), and the metal layer switch procedure at the source and target bend points.

2. **Output:** Minimum cost \(c^e(P_s, P_t, l_t)\) for the candidate bend point pair \((P_s, P_t, l_t)\).

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

\[
c^e(P_s, P_t, l_t) = \min_{0 \leq c_{bc}(P_s, l_t) < l_t, l_t < l_t} \left\{ c_{bc}(P_s, l_t) + c(P_s, B_{ls}^i, B_{lt}^j, T_{lt}) \right\}.
\]  

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:

\[
c^e(P_s, P_t, l_t) = \min_{1 \leq c_{bc}(P_s, l_t) < 0} c^e(P_s, P_t, l_t).
\]

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 \(i\)th candidate bend point pair \((B_{ls}^i, B_{lt}^j)\) is shown in the right part of Fig. 9.

**TABLE II**

**ADDITIONAL NOTATIONS FOR GPU-FRIENDLY Z-SHAPE PATTERN ROUTING**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>((B_{ls}^i, B_{lt}^j))</td>
<td>The (i)th bend point pair of 2D Z-shape patterns.</td>
</tr>
<tr>
<td>(B_{ls}^i)</td>
<td>The bend points on (i)th layer of 3D patterns.</td>
</tr>
<tr>
<td>(c^e(P_s, P_t, l_t))</td>
<td>The minimum cost with ((B_{ls}^i, B_{lt}^j)).</td>
</tr>
<tr>
<td>(c^e(P_s, P_t, l_t))</td>
<td>The minimum cost of (P(P_s)) attached with (P_s \rightarrow T_{lt}).</td>
</tr>
</tbody>
</table>

**Fig. 10.** Overall GPU-friendly 3-D Z-shape pattern routing flow.
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 \(P_s\)'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.

**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 rewrite 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>
<thead>
<tr>
<th>Bench</th>
<th># nets</th>
<th># G-cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>18test5</td>
<td>72394</td>
<td>619×613</td>
</tr>
<tr>
<td>18test8</td>
<td>179863</td>
<td>905×883</td>
</tr>
<tr>
<td>18test10</td>
<td>182000</td>
<td>606×522</td>
</tr>
<tr>
<td>19test7</td>
<td>537577</td>
<td>1053×1011</td>
</tr>
<tr>
<td>19test8</td>
<td>545692</td>
<td>1202×1138</td>
</tr>
<tr>
<td>19test9</td>
<td>895252</td>
<td>1337×1433</td>
</tr>
</tbody>
</table>
TABLE IV
SORTING SCHEMES

<table>
<thead>
<tr>
<th>ID</th>
<th>Sorting Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Descending guide area size</td>
</tr>
<tr>
<td>1</td>
<td>Ascending guide area size</td>
</tr>
<tr>
<td>2</td>
<td>Descending bounding box half perimeter</td>
</tr>
<tr>
<td>3</td>
<td>Ascending bounding box half perimeter</td>
</tr>
<tr>
<td>4</td>
<td>Descending #pins</td>
</tr>
<tr>
<td>5</td>
<td>Ascending #pins</td>
</tr>
</tbody>
</table>

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.

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

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

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:

$$ s = \alpha W + \beta V + \gamma S $$

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

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 FastGRL and FastGRH 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 FastGRH 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 distinctive 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 defined in (15). The FastGR with our proposed GPU-friendly L-shape pattern routing algorithm is named FastGRL [15] and FastGRH is short for our global router integrated with our GPU-friendly hybrid-shape pattern routing algorithm.

Table VII plots the overall performance with total running time and the solution quality on ICCAD2019 benchmarks. By
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× speedup 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 \( \sum_{e \in E}(M_e + N_e) \times L \times L \times L \) 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 \( O(|E|(M + N)L^3) \) 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 \( O(|E|L^2) \) computational amount.

### Table VIII

**BREAKDOWN RUNTIME RESULTS ON ICCAD 2019 BENCHMARKS**

<table>
<thead>
<tr>
<th>Bench</th>
<th>Pattern runtime (s)</th>
<th>CUGR</th>
<th>FastGR</th>
<th>Speedup</th>
<th>FastGR</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1test5</td>
<td>45.111</td>
<td>4.967</td>
<td>9.081×</td>
<td>27.774</td>
<td>1.624×</td>
<td>16.86</td>
</tr>
<tr>
<td>1test5m</td>
<td>7.445</td>
<td>3.314</td>
<td>2.247×</td>
<td>13.084</td>
<td>0.569×</td>
<td>2.320</td>
</tr>
<tr>
<td>1test6</td>
<td>118.117</td>
<td>8.993</td>
<td>13.149×</td>
<td>53.110</td>
<td>2.224×</td>
<td>23.45</td>
</tr>
<tr>
<td>1test6m</td>
<td>18.292</td>
<td>5.715</td>
<td>3.201×</td>
<td>23.690</td>
<td>0.772×</td>
<td>30.08</td>
</tr>
<tr>
<td>1test10</td>
<td>124.533</td>
<td>10.410</td>
<td>11.96×</td>
<td>38.503</td>
<td>3.234×</td>
<td>12.61</td>
</tr>
<tr>
<td>1test10m</td>
<td>18.485</td>
<td>6.788</td>
<td>2.723×</td>
<td>15.858</td>
<td>1.166×</td>
<td>13.42</td>
</tr>
<tr>
<td>1test7</td>
<td>223.947</td>
<td>15.536</td>
<td>14.396×</td>
<td>85.943</td>
<td>2.606×</td>
<td>33.68</td>
</tr>
<tr>
<td>1test7m</td>
<td>34.208</td>
<td>8.658</td>
<td>3.931×</td>
<td>45.032</td>
<td>0.760×</td>
<td>58.73</td>
</tr>
<tr>
<td>1test8</td>
<td>333.965</td>
<td>17.612</td>
<td>18.96×</td>
<td>93.800</td>
<td>3.560×</td>
<td>27.34</td>
</tr>
<tr>
<td>1test8m</td>
<td>51.922</td>
<td>11.982</td>
<td>4.333×</td>
<td>43.210</td>
<td>1.202×</td>
<td>35.98</td>
</tr>
<tr>
<td>1test9</td>
<td>561.337</td>
<td>24.481</td>
<td>22.929×</td>
<td>103.690</td>
<td>5.414×</td>
<td>19.72</td>
</tr>
<tr>
<td>1test9m</td>
<td>88.971</td>
<td>17.976</td>
<td>4.949×</td>
<td>52.054</td>
<td>1.709×</td>
<td>42.957</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAZE runtime (s)</th>
<th>CUGR</th>
<th>FastGR</th>
<th>Speedup</th>
<th>FastGR</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nets to rip up</td>
<td>868</td>
<td>783</td>
<td>1.000</td>
<td>976</td>
<td>0.767</td>
</tr>
<tr>
<td></td>
<td>21.588</td>
<td>5.219</td>
<td>4.136×</td>
<td>2.320</td>
<td>9.304×</td>
</tr>
</tbody>
</table>

### 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 \( O(|E|(M + N)L^3) \) 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 \( O(|E|L^2) \) 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 \( O(|E|(M + N)) \) and \( O(|E|) \), 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 \( O(|E|) \).

On the other hand, the additional merge step for all \((M + N)\) candidates with this structure costs \( O(\log(M + N)) \) 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

<table>
<thead>
<tr>
<th>Bench</th>
<th>Score</th>
<th>Wirelength</th>
<th># Vias</th>
<th># Shorts</th>
<th>Improved (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18test5</td>
<td>16919500</td>
<td>16888000</td>
<td>26988200</td>
<td>26195900</td>
<td>85636</td>
</tr>
<tr>
<td>18test5m</td>
<td>18770900</td>
<td>18701200</td>
<td>27942700</td>
<td>27799600</td>
<td>802385</td>
</tr>
<tr>
<td>18test8</td>
<td>40881100</td>
<td>40822000</td>
<td>64399990</td>
<td>64211700</td>
<td>2174240</td>
</tr>
<tr>
<td>18test8m</td>
<td>42957600</td>
<td>42725000</td>
<td>65546600</td>
<td>64414200</td>
<td>1985740</td>
</tr>
<tr>
<td>18test10</td>
<td>42592100</td>
<td>42579900</td>
<td>66718200</td>
<td>66677100</td>
<td>2308250</td>
</tr>
<tr>
<td>18test10m</td>
<td>44579100</td>
<td>44275300</td>
<td>71071700</td>
<td>71014000</td>
<td>2059820</td>
</tr>
<tr>
<td>19test7</td>
<td>71887100</td>
<td>71763600</td>
<td>11874400</td>
<td>11849500</td>
<td>3128750</td>
</tr>
<tr>
<td>19test7m</td>
<td>67923300</td>
<td>67639200</td>
<td>10711500</td>
<td>10657700</td>
<td>3080660</td>
</tr>
<tr>
<td>19test8</td>
<td>11392900</td>
<td>11384400</td>
<td>18185400</td>
<td>18167800</td>
<td>5750370</td>
</tr>
<tr>
<td>19test8m</td>
<td>11436200</td>
<td>11454900</td>
<td>17763800</td>
<td>17728400</td>
<td>5612590</td>
</tr>
<tr>
<td>19test9</td>
<td>17552300</td>
<td>17536700</td>
<td>27410600</td>
<td>27388400</td>
<td>9603400</td>
</tr>
<tr>
<td>19test9m</td>
<td>17333900</td>
<td>17300200</td>
<td>26778600</td>
<td>26730400</td>
<td>9359980</td>
</tr>
</tbody>
</table>

Average | | | | | 27.855 |

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\textsuperscript{L1}, 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\textsuperscript{H} to get a 27.855% improvement in routing speedup, compared to the CPU-oriented version FastGR. With 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


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.

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/algorithms 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, ICMT 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.