FastGR : Global Routing on CPU-GPU with Heterogeneous Task Graph Scheduler

Siting Liu[1,2], Peiyu Liao[1,2], Rui Zhang[3], Zhitang Chen[4], Wenlong Lv[4], Yibo Lin[1*], Bei Yu[2*]
[1]Peking University [2]Chinese University of Hong Kong
[3]HiSilicon Technologies Co. [4]Huawei Noah's Ark Lab

***Abstract*—Routing is an essential step to integrated circuits (IC) design closure. With the rapid increase of design scales, routing has become the runtime bottleneck in the physical design flow. Thus, accelerating routing becomes a vital and urgent task for IC design automation. This paper proposes a global routing framework running on hybrid CPU-GPU platforms with a heterogeneous task scheduler and a GPU-accelerated pattern routing algorithm. We demonstrate that the task scheduler can lead to 2.307× speedup compared with the widely-adopted batch-based parallelization strategy on CPU and the GPU-accelerated pattern routing algorithm can contribute to 10.877× speedup over the sequential algorithm on CPU. Finally, the combined techniques can achieve 2.426× speedup without quality degradation compared with the state-of-the-art global router.**

I. Introduction

Routing is an important stage in the modern very-large-scale integration (VLSI) design flow. Modern routing is typically divided into global routing and detailed routing. Global routing performs rough routing on a coarse grid graph and generates routing guidance for detailed routing [1]. Detailed routing works on a fine grid graph to connect all wires and minimize design rule violations following the guidance from global routing [2]. In the design flow, global routing also serves as the congestion predictor for proceeding stages like placement [3], [4]. Due to the repeated invocation of global routing, its efficiency is very critical to the design closure.

The essential task in global routing is finding the shortest paths for each net. Due to the large problem scale, modern global routing follows a two-stage procedure, consisting of the pattern routing stage and rip-up and reroute iterations. Pattern routing serves as the initial routing to limits the search space for efficiency [5]. The rip-up and reroute iterations always utilize maze routing to achieve better solution performance [6] by extensive search to find paths for all nets.

Existing global routing algorithms mainly focus on improving the efficiency on CPU [7]–[10], while the speedup is limited due to the threading overhead, limited bandwidth, and cache sizes of CPUs. With the ever-increasing power of GPUs, accelerating global routing on heterogeneous CPU-GPU platforms brings new opportunities for high-performance routing engines.

The literature has extensively explored shortest path searching with GPU [11], [12]. However, most studies only consider the most basic single-source shortest path problem and assume only to find one path on a large graph. This is impractical for routing since we need to route millions of nets subjecting to various objectives and constraints like wirelength, number of vias, and design rules. In this paper, we propose FastGR, a global routing framework that is accelerated on CPU-GPU platforms with a task graph scheduler and GPU-accelerated algorithms. Our scheduler works on the heterogeneous platforms with both CPU and GPU.

The main contributions of this paper are listed as follows,

- We propose a novel GPU-accelerated pattern routing algorithm that can route a batch of nets leveraging the massive parallelism on GPU.
- We propose a high-performance task graph scheduler to distribute CPU and GPU tasks for workload balancing and efficiency.
- Experimental results demonstrate that FastGR can achieve 2.426× overall speedup without any quality degradation compared with the state-of-the-art global router [1]. More specifically, the GPU acceleration can bring 10.877× speedup for pattern routing, and the task scheduler can bring 2.307× speedup for the rip-up and reroute iterations.

The rest of this paper is organized as follows. Section II introduces the problem formulation, the background of modern global routers, and net ordering. Section III presents details of our GPU-based global routing algorithm and the asynchronous task scheduler. Section IV demonstrates the experimental results of our proposed global router. Finally, Section V concludes this paper.

II. Preliminary

A. Problem Formulation

To formulate the global routing problem, we firstly use a set of global routing cells (G-cells) with a group of evenly distributed horizontal and vertical grids to represent the global routing region. A grid graph $G(V, E)$ can be defined by treating each G-cell as a vertex ($v \in V$) and creating an edge ($e \in E$) between every two adjacent G-cells. Global routing is a minimum cost path searching problem on $G(V, E)$. The edge between two G-cells on the same metal layer is called the wire edge, whose capacity is equal to the number of tracks that can go through it, and the routing demand is the number of tracks that need to go through. The edge between two G-cells with the same 2D coordinates but on different metal layers is called the via edge, whose capacity is infinity by many 2D routers while some 3D routers consider the via capacity, *e.g.*, CUGR [1].

B. Modern global router

The most straightforward method for routing is to select a specific net order and then route these nets sequentially in that order. However, the major backward of such a sequential approach is that it suffers from the net ordering and may lead to a poor solution because the earlier routed net might block the routing for its subsequent nets. Modern sequential global routers always follow a two-stage procedure, the pattern routing stage, and rip-up and reroute iterations, with net-ordering to guide. In the most popular framework, it applies

[*]Corresponding authors: byu@cse.cuhk.edu.hk, yibolin@pku.edu.cn

978-3-9819263-6-1/DATE22/©2022 EDAA

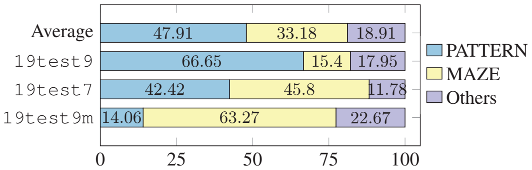


Fig. 1 Runtime breakdown of a modern global router; PATTERN means the pattern routing stage while MAZE means the maze routing stage.

maze routing for rip-up and reroute. Our work also follows this strategy.

The concurrent routing can help to solve the issue caused by net ordering. It will solve all the nets at the same time. The most popular concurrent approach is formulating global routing as a 0-1 integer linear programming problem (0-1 ILP). Even though such an ILP formulation can find the optimal solution when it exists, 0-1 ILP problem is NP-complete. The extremely high time complexity limits the feasible problem size, which cannot tolerate in the industry.

To get an efficient framework, we propose a routing strategy with a practical routing tasks scheduler for asynchronous parallelization. There exists an efficient two-stage sequential routing framework, including pattern routing and maze routing. The pattern routing algorithm gives an efficient routing solution for all the nets. At the same time, maze routing works as rip-up and reroute iterations to reroute the nets with violations for a better solution. In that case, pattern routing stage routes much more nets than maze routing stage.

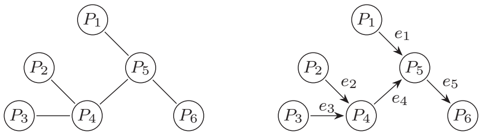
C. Runtime breakdown of a modern global router

To illustrate the percentage of the pattern routing stage and the maze routing iterations in a two-stage router, we select one modern global router with these two stages. The runtime breakdown of this modern global router on three different benchmarks and the average runtime breakdown on ICCAD2019 benchmarks [13] are shown in Fig. 1. Here PATTERN means the runtime percentage of the pattern routing stage, and MAZE refers to the runtime percentage of the maze routing iterations for rip-up and reroute. These three benchmarks include one PATTERN-dominated design, `19test9`, one MAZE-dominated design, `test9m`, and one design with nearly the same percentage of PATTERN and MAZE, `19test7`. Fig. 1 shows that it is PATTERN-dominated on average since the number of nets which pattern routing stage needs to process is much more than the maze routing stage.

D. Net ordering within a single multi-pin net

There are several two-pin nets within one single multi-pin net as shown in Fig. 2(a), we are supposed to determine the net ordering of these two-pin nets since there is the dependency between two connected two-pin nets. One of the most popular methods is to utilize a depth-first search (DFS) traversal to visit all nodes from a random root. Take Fig. 2 as a sample to show this method. All the two-pin nets will perform routing in the reverse order in sequential.

Suppose we choose P_6 as the random root in Fig. 2(a). Then, we perform DFS starting from P_6. DFS traversal visits all the nodes with the order of $P_6, P_5, P_4, P_3, P_2, P_1$. As shown in Fig. 2(b), we label all the two-pin nets in the reverse order e_1, e_2, e_3, e_4, e_5, which will be the order they perform the routing algorithm.



(a) two-pin nets (b) two-pin nets with order

Fig. 2 Sample for ordering.

E. Net ordering in routing

Besides the net ordering scheme within the single multi-pin net, the net ordering among multi-pin nets attracts much more attention. There are several approaches to generate a net sequence for routing, but it is difficult to determine the best. Because an earlier routed net may impede routing for the following net with limited routing resources, net ordering significantly impacts routing solution quality. As a result, a net-ordering strategy for general routing situations is desirable.

Unfortunately, such a universally good scheme is hard to find. Some earlier studies [14] concluded that no single net-ordering strategy could perform better than others in all routing problems. Finding the optimal net ordering scheme has been proven to be NP-hard, which means there is no polynomial-time complexity algorithm to solve this net-ordering problem. Although the net-ordering method may not be the best, several popular net-ordering schemes still exist: (1) The ascending (descending) order of the number of pins within their bounding boxes. If there are more pins inside its bounding box, it is more likely to block other nets in this bounding box. (2) The ascending (descending) order of their wirelength. Routing the shorter nets first always leads to better routability since this kind of net often has less routing flexibility than the longer ones. The other similar metric is the bounding box area.

Researchers always apply the above net-ordering methods in sequential routing [8], [15], [16], but parallelism exists among these multi-pin nets. We utilize this parallelism for acceleration. Therefore, we propose a heuristic task graph scheduler to distribute tasks and then reach parallelization on hybrid platforms.

III. Algorithm

A. Overview

The overall flow of FastGR is shown in Fig. 3. To begin with, we declare a task graph scheduler framework and apply it to both parts in our router, the pattern routing stage and rip-up and reroute iterations, to guide the execution order of different routing tasks. The task graph is constructed from the conflicted relationship between each pair of tasks. Note that a conflict between two routing tasks means that they cannot be processed simultaneously. Our proposed task graph scheduler is used to decide the execution order of all these tasks in the task graph.

We pass the routing tasks obtained from the pattern routing planning strategy into the task graph scheduler in the pattern routing stage. Then we utilize the 3D pattern routing on GPU

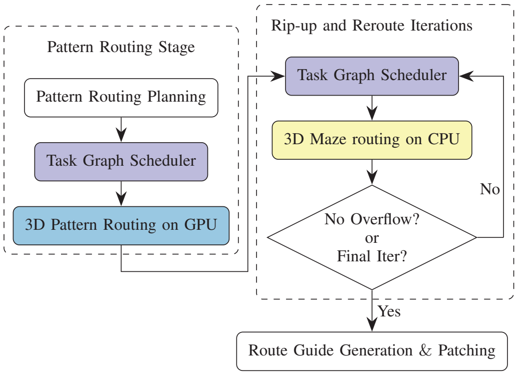


Fig. 3 Overall flow of FastGR.

with the order of the task graph. As for the rip-up and reroute iterations, we get the parallelism among all the maze routing tasks from our task graph scheduler and then call 3D maze routing to maximize the parallelism of the reroute stage on CPU. After several rip-up and reroute iterations, we generate route guides and patches for the detailed routing.

We will discuss the details of our task graph scheduler, our GPU-based 3D pattern routing algorithm, and our task graph generation methods in both two stages in the following sections.

B. Task graph scheduler

We propose a two-stage task graph scheduler to get the order from a set of tasks. The first stage is to construct the task graph from the conflicted relationship between each pair of tasks. Then we determine the execution order for each conflict edge in our task graph to schedule all these tasks. After generating the task graph, we extract one root task batch with the conflict information from this graph. Note that there is no conflict in the root task batch. We split all these tasks into two parts, the root tasks and the non-root tasks. Since there is no conflict within the root tasks, only two conditions exist between each pair of conflicting tasks.

- *One task is in the root batch, and the other is not.* The order is from the task in the root batch to the other.
- *Both the tasks are not in the root batch.* The order is from the task with a smaller task ID to the other. Note that the task ID represents the sorting result of all the tasks.

With this strategy, our task graph scheduler can assign the execution order between each pair of conflicting tasks. Take Fig. 4 as an example. Given a task conflict graph, we pick out an independent root task batch using the batch extraction algorithm described in Algorithm 1. Then we can get the final execution order for the task graph with our task graph scheduler.

C. Pattern routing stage: Task graph generation

We treat a batch of multi-pin nets as one routing task in the pattern routing stage because the number of nets which need to route is quite large. To utilize the parallelism among all the multi-pin nets, we first split the multi-pin nets into several batches with a batch extraction algorithm referring to [2], which is described in Algorithm 1, to maximize the parallelism within batches.

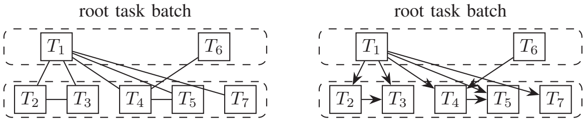


Fig. 4 Sample for task graph scheduler with 7 tasks; The edge in task graph represents the conflict relation between the two connected tasks.

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;
```

Given a set of multi-pin nets `nets`, we first sort all the nets with a sorting strategy, which we will discuss in Section IV-B. Suppose that we sort all the nets with ascending bounding box areas. Then pick out the first nets e (Line 1), the net with the largest bounding box area in the set of remaining nets, into one new empty batch `batch`. After that, we scan the whole set of nets in order and filter out the net which have no conflict with all the nets already in `batch` (Lines 3 to 8). Anytime we find one net that meets this condition, we update the set of remaining nets `nets` and the new batch `batch`. After such a complete scan, we have obtained a batch `batch` with the nearly most number of independent nets. We need to repeat the batch scheduler until the set of remaining nets is empty. Finally, we can obtain a set of nets batches with no conflict in a little overhead.

Since there is no bounding box overlap within each batch, we can simultaneously route the nets in the same batch so that we can treat one batch as one task to construct the task graph in the pattern routing stage. According to the batch extraction algorithm described in Algorithm 1, the task graph we construct from these batches will be a complete graph with edges between every two tasks. Applying our proposed task graph scheduler, we will execute all these tasks in sequential to avoid conflicts.

Fig. 5 illustrates the programming architecture of our GPU-friendly pattern routing for all these routing tasks. Each batch in Fig. 5 represents one routing task, and each task includes several multi-pin nets which need to route. As shown in Fig. 5, we call the 3D pattern routing kernel on the host to solve the routing task. We will assign the kernel to the grid on the device, and many blocks in this grid can be utilized simultaneously with different calculation flows. We use all the threads in one block to process at the same time. Therefore, we apply each block to process one single multi-pin net, and the threads in this block will process the same computation

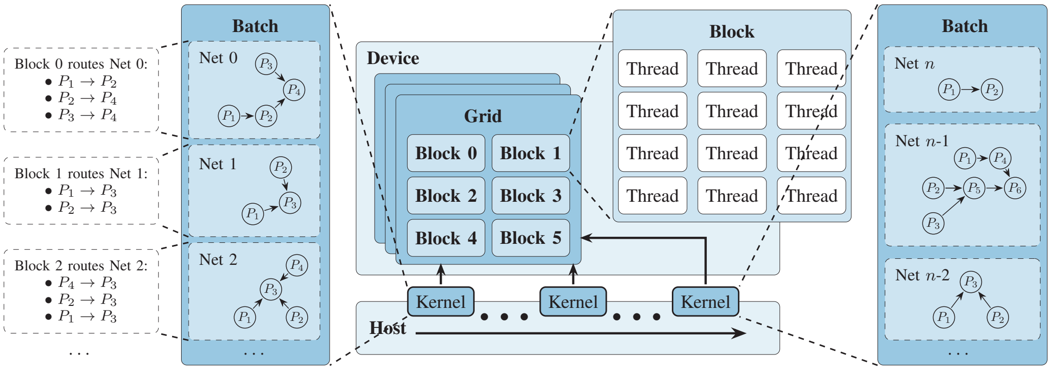


Fig. 5 Programming architecture for the pattern routing stage; On the host, the kernels are called in sequential and each of them is used to process one batch of nets. The nets in the same batch will be routed simultaneously on different blocks on the device.

instructions.

D. Pattern routing stage: GPU-friendly pattern routing

For parallelism, we use several blocks on GPU to process the same number of multi-pin nets in the same batch individually since all the nets can be processed simultaneously, as we discussed before. We will split all n multi-pin nets into several batches and process them in order because their task graph is complete. Then, for each multi-pin net in one block, several two-pin nets should be routed in an order decided by DFS traversal. Pattern routing is a widely-used method to solve the global routing problem for two-pin nets. We reformulate the conventional pattern routing algorithm and utilize GPU computing power, which is demonstrated in Fig. 6.

Take a two-pin net $P_s \rightarrow P_t$ as the example, suppose that the number of metal layer is 4. The left part in Fig. 6 shows one solution of $P_s \rightarrow P_t$ with 3D pattern routing algorithm, and the path solution includes three parts,

- the wire connecting P_s and the bend point B;
- the vias connecting different metal layers through the bend points B and B';
- the wire connecting the bend point B' and P_t.

Besides of these three parts, we need to consider the vias connecting the pin segment, which P_s use to connect the bend point B and the pin segments, which P_s service to connect its child nodes. This kind of vias is used to combine pair of two-pin nets in the same multi-pin net.

Let L be the number of metal layers. The 3D L-shaped pattern routing has $L \times L$ layer combinations, although some combinations are invalid because of the routing direction constraint on each metal layer. We reformulate the L-shape pattern routing with layer assignment into a four-stage graph computation problem to enumerate all these $L \times L$ combinations simultaneously, which is more friendly to GPU.

Let $\boldsymbol{c}^{(i)}$ be the cost vector at the end of the i^{th} step, where $\boldsymbol{c}^{(0)}$ is simply the zero vector to initialize the cost. Additionally, let $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \boldsymbol{w}^{(4)}$ be the weights vector in step 1, 2, 4 and $\mathbf{W}^{(3)}$ be the weight matrix in step 3.

Referring to the cost scheme in [1], we use $\text{cost}_{\text{wire}}(u, v)$ to represent the cost of a wire edge, which includes wirelength cost and congestion cost, and $\text{cost}_{\text{via}}(u, u')$ to mean the cost of a via edge, where u and u' are the lower and upper G-cells connected by a via and v is on the same metal layer with u.

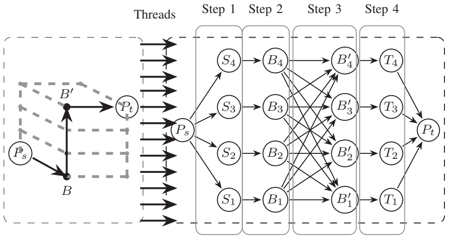


Fig. 6 GPU-friendly 3D pattern routing; The whole flow is split into four steps and the number of metal layer in this sample is 4. The step 1, 2, 4 are the vector addition operation while the step 3 is the addition-minimum operation.

The first step is to calculate the via cost of connecting the pin segment of P_s and all the children segments of P_s. The nodes in the first step are represented as S_l, $l \in \{1, 2, 3, 4\}$, where l means the index of metal layer. The edge between P_s and S_l represents that the pin segment of P_s, which is used to connect with the bend point B, is assigned to l^{th} layer. The weight of this edge is the via cost to connect this pin segment with all the children segments of P_s. The formal formulation of l^{th} entry of the edge weights vector $\boldsymbol{w}^{(1)}$ is as follows,

$$w_l^{(1)} = \text{cost}_{\text{via}}(S_l, P_{sc}),\ 0 < l \leq L, \tag{1}$$

where P_{sc} means all the children nodes of P_s.

The second step is the wire cost connecting the pin segment of P_s and the bend point B. B_l refers to the bend point connecting to P_s in l^{th} layer. The weight of edge S_l to B_l is the wire cost in l^{th} layer to connect these two points. The formal formulation of l^{th} entry of the edge weights vector $\boldsymbol{w}^{(2)}$ is as follows,

$$w_l^{(2)} = \text{cost}_{\text{wire}}(S_l, B_l),\ 0 < l \leq L. \tag{2}$$

The third step works as the layer assignment. The connection between B_{l_1} and $B^{'}_{l_2}$ represents the combination of the layer l_1 from the source and the layer l_2 to the target. The cost of this edge is the via cost to connect l_1 layer and l_2

TABLE I Sorting strategies.

ID	Sorting Strategy
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

layer. The entry of the edge weights matrix $\mathbf{W}^{(3)}$ at the ${l_1}^{\text{th}}$ row and the ${l_2}^{\text{th}}$ column is

$$w^{(3)}_{l_1,l_2} = \text{cost}_{\text{via}}(B_{l_1}, B'_{l_2}),\ 0 < l_1 \leq L\ ,\ 0 < l_2 \leq L. \quad (3)$$

The forth step is used to calculate the wire cost between the bend point B' and the pin segment of P_t. T_l represents the pin segment of P_t, which is used to connect to the bend point B'_l, is assigned into l^{th} layer. The edge weight between B'_l and T_l is the wire cost between them. The formal formulation of l^{th} entry of the edge weights vector $\boldsymbol{w}^{(4)}$ is as follows,

$$w^{(4)}_l = \text{cost}_{\text{wire}}(B'_l, T_l),\ 0 < l \leq L. \quad (4)$$

The vector computation for step 1, 2, 4 are addition operation and the formulation is as follows,

$$\boldsymbol{c}^{(i)} = \boldsymbol{c}^{(i-1)} + \boldsymbol{w}^{(i)}, i \in \{1,\ 2,\ 4\}. \quad (5)$$

The computation in step 3 is the addition-minimum operation since we can get the best layer combinations in this layer assignment step. The formal formulation is shown in Equation (6),

$$c^{(3)}_{l_2} = \min_{0<l_1\leq L} \left\{ c^{(2)}_{l_1} + w^{(3)}_{l_1,l_2} \right\}, 0 < l_2 \leq L. \quad (6)$$

E. Parallel rip-up and reroute iterations

The pattern routing stage always cannot get a violation-free routing solution for many multi-pin nets. Several rip-up and reroute iterations should be conducted to reduce the overall number of violations. We only need to reroute the nets with violations in our proposed rip-up and reroute method to save running time. We treat each multi-pin net as one routing task, which differs from the pattern routing stage. The number of multi-pin nets in the pattern routing stage is more significant than the rip-up and reroute iterations. Then, we apply our task graph scheduler to these routing tasks with their conflict relationship to get the ordered task graph to guide the execution of the nets which need to reroute.

Therefore, with the ordered task graph we generated from our task graph scheduler, we can quickly maximize the parallelism of rip-up and reroute iteration using Taskflow [17]. Taskflow is a C++ tasking toolkit that automatically executes parallel programs with the task dependency graph. It can utilize our ordered task graph as the task dependency graph and perform the tasks in maximum parallelism using CPU threads.

IV. Experimental Results

A. Experimental Setup

We implemented our proposed task graph scheduler and GPU-friendly pattern routing algorithm on the pattern routing stage of CUGR [1] and integrated the scheduler into the rip-up and reroute iterations. We evaluate the results using ICCAD2019 benchmarks [13]. We implement all the algorithms in C++/CUDA and conduct the experiments on a 64-bit Linux machine with Intel Xeon 2.2 GHz CPU and one GeForce RTX 2080 GPU.

TABLE II Experimental results of different sorting strategies; The sorting strategies are only applied to the rip-up and reroute iterations with maze routing.

Benchmarks	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

B. Sorting Strategy

The experimental results with different sorting strategies show that the net ordering affects the final solution quality and the running time. Since the solution of the pattern routing stage will affect rip-up and reroute iterations, we choose six different strategies only applied to the rip-up and reroute iterations to show the effect of net ordering, and TABLE II demonstrates the experimental results. With different sorting strategies, the rip-up and reroute iterations' running time is different because the routing order and the number of nets with violation affect the solution quality.

Furthermore, we apply a score, which considers three parts: wirelength, vias, and shorts, to reflect the solution quality. The computation formulation of score s is shown in Equation (7),

$$s = \alpha W + \beta V + \gamma S, \quad (7)$$

where W represents the wirelength, V means the number of vias, and S is the number of shorts violations. In addition, α, β, γ are three weights of wirelength, vias number, and shorts number for the score weighted sum calculation, respectively. Note that in our experiments, we set α to 0.5, β to 4, and γ to 500 as the same as the setting in ICCAD2019 contest.

To be more comprehensive, we select two benchmarks to evaluate the effect of routing order, where `18test10` has nine metal layers, and `18test10m` has only five metal layers. In TABLE II, we use TOTAL to represent total running time; We use PATTERN to mean the running time of the pattern routing part, while MAZE to describe the runtime of maze routing iterations. As shown in the experimental results TABLE II, for both these two benchmarks, considering bounding box half perimeter as the metric to sort nets can have a better effect on the solution quality, which is also related to the running time.

C. Overall Acceleration

We conducted experiments on 12 different benchmarks of the ICCAD2019 contest in advanced nodes. Half of them, which end with "`m`", have only five metal layers, while the other six have nine metal layers. Furthermore, with the discussion in Section IV-B, we finally order all the routing tasks in both two stages in ascending bounding box half

TABLE III Experiment results on ICCAD 2019 benchmarks.

Benchmarks	Total runtime (s)			Pattern routing runtime (s)			Maze routing runtime (s)			Score		
	CUGR	Ours	Speedup	CUGR	Ours	Speedup	CUGR	Ours	Speedup	CUGR	Ours	Ratio
`18test5`	80.645	24.130	3.342×	45.111	4.516	9.988×	21.588	5.640	3.828×	16931900	16924090	0.9995
`18test5m`	83.49	34.640	2.410×	7.445	2.667	2.791×	63.32	18.409	3.440×	18760300	18800190	1.0021
`18test8`	260.982	99.609	2.620×	118.117	7.969	14.822×	108.986	57.745	1.887×	40870500	40880730	1.0003
`18test8m`	250.26	134.991	1.854×	18.292	4.866	3.759×	201.58	88.408	2.280×	42924700	42798740	0.9971
`18test10`	354.347	115.248	3.075×	124.533	9.416	13.226×	199.777	73.915	2.703×	42622300	42585330	0.9991
`18test10m`	339.471	170.716	1.989×	18.485	5.780	3.198×	291.28	113.259	2.572×	44193400	44448860	1.0058
`19test7`	527.955	192.480	2.743×	223.947	13.292	16.849×	241.787	111.138	2.176×	71855400	71884100	1.0005
`19test7m`	340.489	191.982	1.774×	34.208	7.307	4.681×	244.686	118.459	2.066×	67958400	68015290	1.0008
`19test8`	529.193	173.186	3.056×	333.965	15.351	21.756×	96.888	53.399	1.814×	113946000	113940700	1.0000
`19test8m`	520.547	332.662	1.565×	51.922	9.598	5.410×	371.748	215.034	1.729×	114748000	114854000	1.0009
`19test9`	842.18	255.149	3.301×	561.337	20.463	27.432×	129.707	74.663	1.737×	175429000	175499000	1.0004
`19test9m`	632.577	458.874	1.379×	88.971	13.457	6.612×	400.227	275.312	1.454×	173154000	173252000	1.0006
Average			**2.426×**			**10.877×**			**2.307×**			**1.000**

perimeter to achieve better performance of the running time and the solution quality.

We evaluate the total runtime, pattern routing runtime, and the runtime of maze routing iterations to demonstrate the acceleration performance of our proposed task graph scheduler and our GPU-friendly pattern routing algorithm.

As shown in TABLE III, the task scheduler can contribute to 2.307× speedup over the widely-adopted batch-based parallelization strategy on CPU. To shorten the data transmission running time, we apply the zero-copy technique in our implementation, which can help us limit the data-passing time in 1s for all these benchmarks. With the zero-copy technique, our GPU-friendly pattern routing algorithm can contribute 10.877× speedup over the sequential algorithm on CPU. The combined techniques can achieve 2.426× speedup with a comparable solution quality to the state-of-the-art global router.

Further analysis on pattern routing runtime reveals that the performance of our GPU-friendly pattern routing algorithm is strengthened when the case has a larger design scale but weakened when the metal layer number is small. Modern circuits, on the other hand, have much more metal layers than only five, thus our GPU-friendly pattern routing is desirable in the industry.

V. Conclusion

In this paper, we propose a global routing framework, FastGR, running on hybrid CPU-GPU platforms with a heterogeneous task graph scheduler and a GPU-accelerated pattern routing algorithm. Evaluation on ICCAD2019 benchmarks demonstrates that our task graph scheduler alone can contribute 2.307× speedup on CPU. Applying both our GPU-friendly pattern routing and the task graph scheduler can lead to 10.877× speedup on GPU. Overall, we can achieve up to 2.426× speedup with comparable solution quality results compared with the state-of-the-art global router. The results of this paper highlight the vital role of GPU-based kernel algorithms and the task scheduler in routing. Both of them can assist to reduce design cycles. Finally, we should keep a close eye on the development of the task graph scheduler and the utilization of GPUs power in the VLSI design flow.

Acknowledgement

This work is supported by The Research Grants Council of Hong Kong SAR (No. CUHK14209420), The National Natural Science Foundation of China (No. 62004006), and HiSilicon.

References

[1] 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.

[2] 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 TCAD*, vol. 39, no. 9, pp. 1902–1915, 2019.

[3] X. He, T. Huang, W.-K. Chow, J. Kuang, K.-C. Lam, W. Cai, and E. F. Young, "Ripple 2.0: High quality routability-driven placement via global router integration," in *Proc. DAC*, 2013, pp. 1–6.

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

[5] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh, "Pattern routing: Use and theory for increasing predictability and avoiding coupling," *IEEE TCAD*, vol. 21, no. 7, pp. 777–790, 2002.

[6] C. Y. Lee, "An algorithm for path connections and its applications," *IRE transactions on electronic computers*, no. 3, pp. 346–365, 1961.

[7] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "FastRoute: An efficient and high-quality global router," *Proc. VLSI Design*, vol. 2012, 2012.

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

[9] M. D. Moffitt, "Maizerouter: Engineering an effective global router," *IEEE TCAD*, vol. 27, no. 11, pp. 2017–2026, 2008.

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

[11] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier, "All-Pairs Shortest Path algorithms for planar graph for GPU-accelerated clusters," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 91–103, 2015.

[12] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "Phast: Hardware-accelerated shortest path trees," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 940–952, 2013.

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

[14] L. C. Abel, "On the ordering of connections for automatic wire routing," *IEEE Transactions on Computers*, vol. 100, no. 11, pp. 1227–1233, 1972.

[15] 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.

[16] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "Nctu-gr 2.0: Multithreaded collision-aware global routing with bounded-length maze routing," *IEEE TCAD*, vol. 32, no. 5, pp. 709–722, 2013.

[17] 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.