GAMER: GPU-Accelerated Maze Routing

Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, *Senior Member, IEEE*, and Martin D. F. Wong, *Fellow, IEEE*

***Abstract*—Maze routing is usually the most time-consuming step in global routing and detailed routing. A commonly used maze routing method is to start from one pin and iteratively connect the current route to the closest unconnected pin. This method reduces the maze routing problem to multiple multisource–multidestination shortest path problems. The shortest path problem in VLSI routing has: 1) rectilinear routing directions and 2) preferably small via usage. By utilizing these two characteristics, we propose a novel parallel algorithm called GAMER to accelerate the multisource–multidestination shortest path problem for VLSI routing. GAMER decomposes the shortest path search into alternating vertical and horizontal *sweep* operations, and two parallel algorithms are proposed to accelerate a *sweep* operation from $O(n^2)$ to $O(\log_2 n)$ on a grid graph of $n \times n$. Several techniques of applying GAMER on irregular routing regions are also introduced. Experiments are conducted by integrating GAMER into the state-of-the-art academic global router CUGR. CUGR adopts a two-level maze routing scheme, including coarse-grained routing and fine-grained routing, and they can be accelerated by 19.85× and 2.59×, respectively, with GAMER, achieving an overall speedup of 2.7× without quality degradation.**

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

I. Introduction

GLOBAL routing is a crucial part in today's electronic design automation (EDA) tool chain. It is commonly used to generate route guides for detailed routers to accelerate the process and to avoid potential congestion. Besides, it is also useful for the early estimation of wire length and congestion during the placement stage [1], [2]. Therefore, the industry's and academia's crave for faster global routing algorithms has never stopped.

For most global routers and detailed routers, maze routing is the most time-consuming step. Many of them adopt the negotiation-based rip-up and reroute method introduced in [3]. Hard-to-route nets are ripped-up and rerouted many times with incrementally changing history cost until getting a feasible solution. Many techniques are proposed to speed up maze routing. The simplest one is to limit the search space of the router. For example, FastRoute 2.0 [4] proposed monotonic routing to limit the exploration to paths getting monotonically closer to the destination. The bounded-length maze routing proposed in [5] will predetermine the maximum distance to explore and save time by not exploring farther regions. This type of methods brings about limited improvement to running time, and all sacrifice the routing quality to some extent.

Another kind of approach makes use of the multicore architecture of modern computers and goes multithreading. One way to do this is to separate nets by their bounding boxes and create a task pool. Each thread will search for a net in the task pool whose bounding box does not overlap with any other nets being routed at the moment and perform maze routing [6]. However, if the bounding boxes are too big, the level of parallelism for this method is low. The approach described in [7] attempts to solve this problem by allowing nets with overlapping bounding boxes to be routed together, and fix any possible overflows afterward by rerouting. SPRoute [8] does not forbid routing in the same region if and only if the region has abundant routing resources. NCTU-GR 2.0 [9] also allows nets with overlapping bounding boxes to be routed simultaneously, but they adopt a more sophisticated technique to avoid the racing situation. A heuristic is developed to mark the resources that have high chance of collision and extra cost will be induced upon using these marked resources.

The approaches mentioned above all achieve net-level parallelism to some degree. However, some extra efforts are needed to resolve data racing, which may lead to unbalanced workloads and routing performance degradation. Besides, as technology evolves over time, graphics processing units (GPUs) are standing out, and can provide better solution to parallelism. There are relatively fewer attempts to load maze routing onto GPUs. The router proposed in [10] making use of a GPU implementation of the A* algorithm has easily achieved around 3× speedup compared to a CPU router. However, the quality has degraded severely and the wirelength of their method increases by around 2.5%. Besides, their method cannot be applied to multipin nets without breaking them into two-pin nets.

In this work, we propose a GPU-friendly algorithm for maze routing based on a novel *sweep* operation. This operation sweeps each row and column of the routing space alternately to update the routing cost of each grid cell (G-cell). The sweeps of each row or column in an iteration are independent and, thus, can be fully parallelized, while a single row or column sweep can also be parallelized to achieve a logarithmic running time. We incorporate our GPU maze router to the state-of-the-art opensourced LEF/DEF-based global router CUGR [11], and use it to accelerate both the coarse-grained maze routing

Manuscript received 11 January 2022; revised 9 April 2022; accepted 26 May 2022. Date of publication 17 June 2022; date of current version 20 January 2023. This work was supported in part by the Grant from the AI Chip Center for Emerging Smart Systems Ltd. This article was recommended by Associate Editor P. Gupta. *(Corresponding author: Shiju Lin.)*

The authors are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: sjlin@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2022.3184281

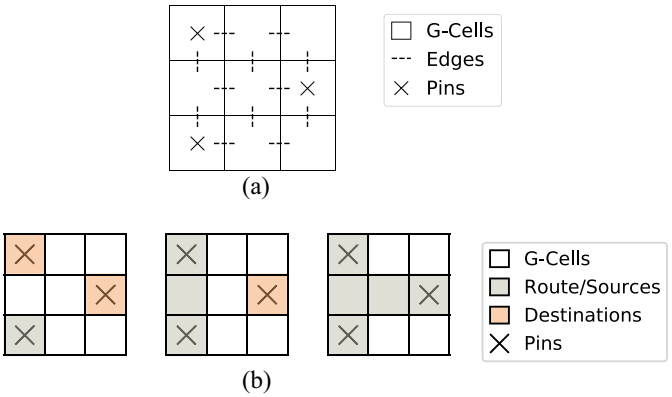


Fig. 1. (a) Grid graph and (b) routing.

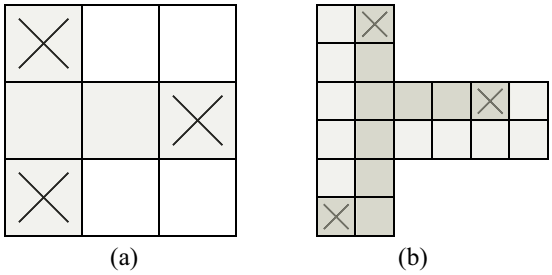


Fig. 2. Two-level maze routing. (a) Coarse-grained level. (b) Fine-grained level.

and the fine-grained maze routing to show the effectiveness of our algorithm. The main contributions of this work are as follows.

1) A new maze routing scheme via a novel *sweep* operation is proposed. It is both efficient and effective (optimal), given that via usage in VLSI routing is discouraged.
2) Two massively parallel algorithms are developed for the *sweep* operation, which are easy to understand and implement.
3) Several useful techniques for routing on irregular regions (rectilinear polygons) are introduced.
4) GAMER is integrated into the state-of-the-art academic global router CUGR. Experiments show that GAMER achieves 19.85× speedup on the coarse-grained maze routing (rectangular routing regions) and 2.59× speedup on the fine-grained maze routing (rectilinear routing regions), leading to an overall 2.7× speedup over CUGR.

The remainder of this article is organized as follows. Section II introduces the background of maze routing and a state-of-the-art global router CUGR. In Section III, a new formulation and two novel parallel algorithms for maze routing are proposed. The parallel maze routing is further extended to handle nonrectangular routing regions in Section IV. In Section V, some technical details are discussed. Finally, experiments are shown in Section VI and this article is concluded in Section VII.

II. Preliminaries

A. Sequential Maze Routing

In maze routing, the routing region is represented by a grid graph with edges (wires) between adjacent grids (G-cells), as shown in Fig. 1(a). Each wire has a congestion cost (wire cost), which is an estimation of congestion or some other relevant routing costs. The distance of a path to a G-cell is the sum of the costs of all the wires on that path. A net contains a number of pins, each of which locates in one grid. The goal of maze routing is to find a route to connect all the pins in a net with the smallest cost. Typical maze routers use shortest path algorithms or A^* search. They start from an arbitrary pin, find the shortest path to the unconnected pins, and connect the one with the shortest distance to the current route. Fig. 1(b) shows an example of this routing process on a net with three pins. If the routing starts from the lower left pin, it performs a shortest path search to find out the shortest paths to the two unconnected pins, and connects to the closer one (upper left pin). Next, using all the G-cells in the current route as sources, the shortest path search is performed again and the last pin is connected.

After finding out which pin P to connect, we need to trace back and generate a subpath to be connected to the current route R. We start from the destination pin P and iteratively move from the current position to a neighboring G-cell until reaching a source on route R.

B. CUGR

CUGR is an opensource global router developed from the ICCAD 2019 contest. It directly operates on a 3-D routing region. A 3-D routing region is represented by multiple layers of 2-D grid graphs. Each layer has a routing direction, and all nets in a layer need to be routed in the dedicated direction. Different layers are connected by vias. In CUGR, 3-D pattern routing with layer assignment is first used to generate routes for all nets. Since the initial routing results may have many congestion violations, it will go through several iterations of rip-up and reroute. In the reroute stage, a two-level maze routing scheme is used. In the coarse-grained maze routing, a block of G-cells (5×5 in CUGR) is merged to form one coarse-grained G-cell. 3-D maze routing is then performed on this coarse-grained grid graph. With the maze routing result on this coarse-grained grid graph as route guides, fine-grained maze routing will be performed to route nets within the guide. An example of this two-level 2-D routing is shown in Fig. 2. The coarse-grained and fine-grained routing are shown, respectively, in Fig. 2(a) and (b). In this example, 2×2 G-cells are merged to one coarse-grained G-cell.

III. Highly Parallel Maze Routing

A. Novel SWEEP Operation for Maze Routing

As discussed in Section II-A, maze routing can be regarded as a multisource–multidestination problem on a grid graph. We propose an operation called *sweep*, which updates the shortest distances to all G-cells with edges in one direction. Horizontal (vertical) sweeps only use horizontal (vertical) edges. By alternating horizontal and vertical sweeps many times, the shortest

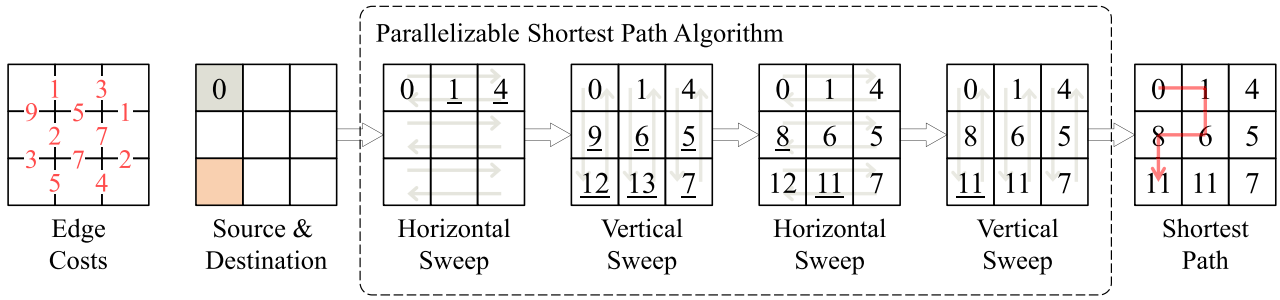


Fig. 3. Shortest path via alternating sweeps.

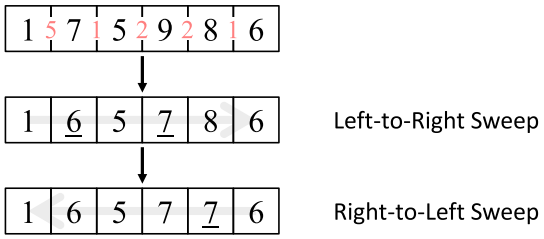


Fig. 4. Example of a row sweep.

distance to every G-cell can be correctly found. An example of applying sweep operations in maze routing is shown in Fig. 3. The red numbers and the black numbers are the edge costs and the current shortest distances, respectively. Some grids are empty because they are currently not reachable. The shortest path from the upper left grid to the lower left grid is indicated in the last grid graph. The shortest path has horizontal, vertical, horizontal, and vertical directions in order. If four sweeps in the same ordered directions are performed as indicated by a dashed rectangle, the shortest path can be correctly found. Note that directional changes (turning points in the route) in VLSI routing are realized with vias, which are usually discouraged for better timing, yield, and reliability. This characteristic lowers the number of directional changes in routing paths, and the number of alternating sweeps needed in practice is thus small.

Since horizontal and vertical sweeps are identical except for the direction, the following discussion will be focused on horizontal sweeps. A *row sweep* only considers one single row, so a horizontal sweep in Fig. 3 consists of three row sweeps. In a horizontal sweep, different row sweeps have no dependencies, so they can work in parallel (interrow parallelism). In the following, the sweep operation will be formulated, and two algorithms that can efficiently compute a row sweep in $O(\log_2 n)$ on GPU will be introduced (intrarow parallelism). Combining both the interrow and the intrarow parallelisms, a horizontal sweep can be efficiently computed in $O(\log_2 n)$.

B. Formulation

Suppose we have n G-cells numbered $0, \ldots, n-1$ in a row and let d_i be the current shortest distance to G-cell i and c_i be the wire cost from G-cell $i-1$ to G-cell i, a row sweep is to update the shortest distances to the G-cells using wires in this row. An example of a row sweep is shown in Fig. 4.

Algorithm 1 $O(n)$ Dynamic Programming Sweep

d_i : current shortest distance to G-cell i
c_i : edge cost between G-cell $i-1$ and G-cell i
1: $d_0^* \leftarrow d_0$
2: **for** $i \leftarrow 1$ to $n-1$ **do**
3: $\quad d_i^* \leftarrow min\{d_i, d_{i-1}^* + c_i\}$
4: **end for**

The red numbers are edge costs, and the black numbers are the current shortest distances to the G-cells. An update of the shortest distance is indicated by an underline. A row sweep can be formulated as

$$d_i^* = \min_{0 \le j \le i} \left(d_j + \sum_{k=j+1}^{i} c_k \right). \tag{1}$$

d_i^* denotes the new shortest distance. Note that (1) is only for one direction (left to right), and the other direction (right to left) can be similarly derived. A simple but efficient way to compute (1) is to use dynamic programming as shown in Algorithm 1. G-cells are iterated from left to right, and d_i is updated using d_{i-1} and c_i. Note that the order of visiting G-cells (left to right) must be followed for correctness, so it cannot be trivially parallelized. In the left to right sweep in Fig. 4, the second and fourth G-cells are updated because $1+5=6<7$ and $5+2=7<9$.

C. Parallelization With Conditional Partial Sum

An example of how to parallelize a row sweep is shown in Fig. 5. It can be viewed as an iterative method looking from top to bottom or as a divide-and-conquer method looking from bottom to top. The divide-and-conquer perspective is used for easier explanation. A row of G-cells is first divided into two halves of the same size, and then two row sweeps are recursively performed on both halves. Finally, we need to combine both halves to form a complete row sweep. In this step, G-cells on the left half should be used to update those on the right. In fact, it is only needed to use the last G-cell on the left half to update every G-cell on the right, because any path from the left half to the right half has to pass through the last G-cell on the left. Moreover, the recursive row sweep guarantees that the last G-cell is already fully updated by all the G-cells in the left. Note that the operations in each iteration have no dependencies and can work in parallel.

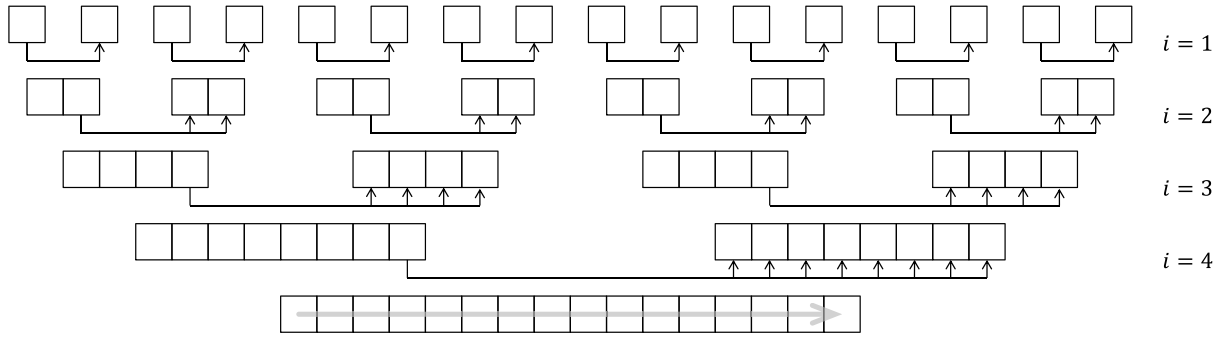


Fig. 5. Parallelization of a row sweep.

Algorithm 2 Parallel Sweep With Conditional Partial Sum

d_i^0 : initial shortest distance to G-cell i
c_i^0 : edge cost between G-cell $i-1$ and G-cell i
1: **for** $i \leftarrow 1$ to $\log_2 n$ **do** ▷ i in Figure 5
2: **for** $j \leftarrow 0$ to $n-1$ **in parallel do**
3: **if** $j \mod 2^i \geq 2^{i-1}$ **then**
4: $idx \leftarrow \lfloor \frac{j}{2^{i-1}} \rfloor \cdot 2^{i-1} - 1$
5: $d_j^i \leftarrow \min\left\{d_j^{i-1}, c_j^{i-1} + d_{idx}^{i-1}\right\}$
6: $c_j^i \leftarrow c_j^{i-1} + c_{idx}^{i-1}$
7: **end if**
8: **end for**
9: **end for**

Algorithm 3 Work-Efficient Parallel Prefix Sum

1: **procedure** PREFIXSUM($x_1, ..., x_n$)
2: **for** $i \leftarrow 1$ to $\frac{n}{2}$ **in parallel do**
3: $y_i \leftarrow x_{2\cdot i} + x_{2\cdot i+1}$
4: **end for**
5: PREFIXSUM($y_1, ..., y_{\frac{n}{2}}$)
6: **for** $i \leftarrow 1$ to $\frac{n}{2}$ **in parallel do**
7: $x_{2\cdot i} \leftarrow y_i$
8: $x_{2\cdot i+1} \leftarrow y_{i-1} + x_{2\cdot i+1}$
9: **end for**
10: **end procedure**

The detailed algorithm is shown in Algorithm 2. One difficulty in this parallelized row sweep might be using the correct accumulated edge cost for each update. A simple but efficient method of dynamically computing the conditional partial sum c_i is proposed and used in Algorithm 2. Note that i in Fig. 5 corresponds to the i in line 1 of Algorithm 2. If j defined in line 2 satisfies the if-condition in line 3, G-cell j is on the right half of the current row sweep and should be updated. idx in line 4 is the last G-cell on the left half, which will be used to update G-cell j. The inner loop of Algorithm 2 can be parallelized because all operations inside it have no dependencies. This algorithm can reduce the running time from $O(n)$ (Algorithm 1) to $O(\log_2 n)$ if enough computing resources are provided. The total work done by this method is $O(n \log_2 n)$, because $O(n)$ work is needed for each of the $\log_2 n$ iterations. This algorithm is thus work inefficient because it will perform asymptotically more work (a logrithmic factor) than required sequentially.

D. Reformulation and Work-Efficient Algorithm

We can further optimize the parallel sweep algorithm by reformulating (1) as a prefix sum problem and a prefix min problem. The reformulation allows us to effortlessly obtain good parallel algorithms for sweeps by applying the existing parallel prefix sum algorithms.

Defining $s_i = \sum_{j=0}^{i} c_j$, we can use it to replace the summation in (1)

$$d_i^* = \min_{0 \leq j \leq i} \left(d_j + \sum_{k=j+1}^{i} c_k \right) = \min_{0 \leq j \leq i} (d_j + s_i - s_j). \tag{2}$$

We can further derive (3) by subtracting s_i on both sides of (2)

$$d_i^* - s_i = \min_{0 \leq j \leq i} (d_j - s_j). \tag{3}$$

Equation (3) is a prefix min problem, which is similar to the prefix sum problem but with the operator *min* instead of $+$. The parallelization of the prefix sum problem has been well studied [12], and it has been generalized to handle not just $+$ but other binary associative operators as well. Operator *min* is a binary associative operator, so we can utilize the best parallel prefix sum algorithm for computing (3).

With (3), a row sweep can be decomposed into the following.

1) *Prefix Sum Problem:* The prefix sum s_i of c_i.
2) *Prefix Min Problem:* The prefix min of $d_i - s_i$.

A work-efficient parallel algorithm can be found at [13]. The detailed algorithm and an example are shown in Algorithm 3 and Fig. 6, respectively. Computing the prefix sum $s_1, \ldots, s_n$ of $x_1, \ldots, x_n$ has the following steps.

1) Pair consecutive items and compute the sum of each pair: $y_i = x_{2\cdot i} + x_{2\cdot i+1}$.
2) Recursively compute the prefix sum $z_1, \ldots, z_{n/2}$ of $y_1, \ldots, y_{n/2}$.
3) Express each s_i as the sum of at most two terms of x_i and z_i.

This algorithm decomposes a prefix sum problem of size n into another prefix sum problem of size $n/2$ (step 2) plus $O(n)$ computation (steps 1 and 3), so the total work of this algorithm is

$$\sum_{i=0}^{\log_2 n} O\left(\frac{n}{2^i}\right) = O(n).$$

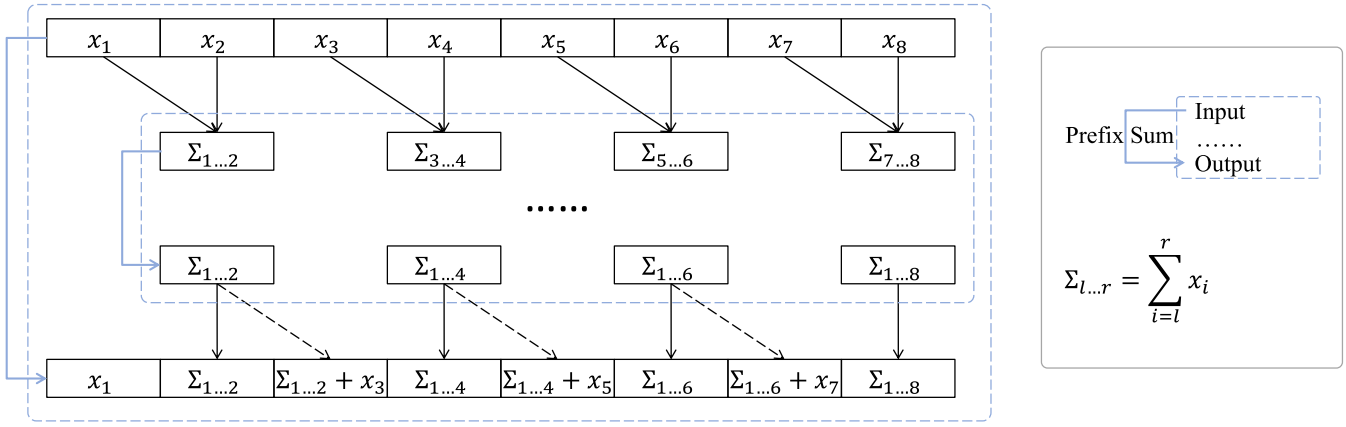


Fig. 6. Work-efficient parallel algorithm for prefix sum.

It shows that Algorithm 3 is work efficient because it performs the same asymptotic amount of work as that of the sequential algorithm (Algorithm 1).

IV. Routing on Rectilinear Polygons

A. Irregular Routing Regions

Global routing is usually conducted on a regular grid graph. However, irregular routing regions are common in detailed routing, because detailed routing relies on the route guides generated by global routing to reduce the solution space, and most route guides are rectilinear polygons. In global routers with the two-level maze routing scheme, the relationship between the coarse-grained routing and the fine-grained routing is similar to that between global routing and detailed routing, so the fine-grained routing also has irregular routing regions defined by the route guides. An example is shown in Fig. 2. The fine-grained routing region consists of five coarse-grained G-cells.

In the following, $\langle\langle N, M\rangle\rangle$ is used to denote a CUDA kernel with N blocks, each of which has M threads. The configuration of a kernel can be represented in this way because every block in a CUDA kernel must have the same number of threads. This feature makes it difficult to apply GAMER in nonrectangular routing regions. For a rectangular routing region of N rows and M columns, a horizontal sweep can be done by an $\langle\langle N, M\rangle\rangle$ kernel. Each row is a block running Algorithm 2 and different rows are independent of each other. Similarly, a vertical sweep can be done by an $\langle\langle M, N\rangle\rangle$ kernel. However, different rows or different columns may have different numbers of G-cells on an irregular rectilinear region. In order to handle this problem, we propose the following method to map a nonrectangular routing region to a rectangular one by extracting routing tracks and placing them onto a rectangle.

B. Routing Track Extraction and Placement

We will assume the routing direction to be horizontal in this section, and the vertical one can be similarly derived. A routing track refers to a row of G-cells within the routing region. The main idea of mapping a nonrectangular region to a rectangular one is to reorganize routing tracks to form a rectangular shape. The procedure and an example are shown in Algorithm 4 and Fig. 7, respectively. First, the number of threads per block is determined, which is also the number of columns in the final rectangle (5 in Fig. 7 and C in Algorithm 4). C must be no less than the length of the longest routing track. If C is larger than 1024 (the maximum number of threads in a CUDA block), the CUDA kernel should be refined by assigning more work to every thread. For example, each of the 1024 threads needs to work twice as much when C is 2048. Next, the routing tracks are extracted and placed row by row inside the rectangle. If the current row cannot accommodate a whole routing track,

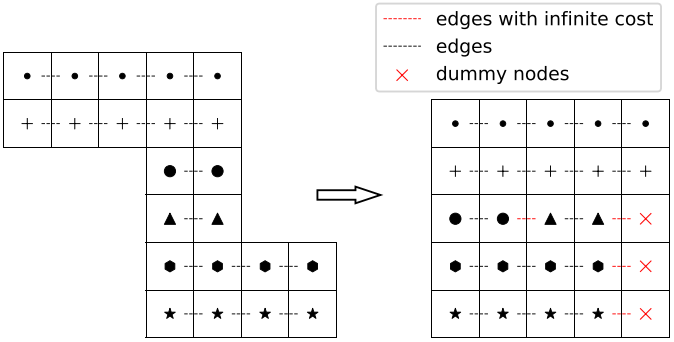


Fig. 7. Rectilinear polygon to rectangle.

Algorithm 4 Mapping Rectilinear Polygon to Rectangle

```
   C : the number of columns  ▷ Columns from 0 to C − 1
1: row ← 0                    ▷ Current Row Number
2: col ← 0                    ▷ Current Column Number
3: for every routing track t do
4:     if col + t.len > C then
5:         fill (row, col → C − 1) with dummy nodes
6:         row ← row + 1
7:         col ← 0
8:     end if
9:     cost of edge (row, col − 1 → col) ← ∞
10:    place t at (row, col → col + t.len − 1)
11:    col ← col + t.len
12: end for
```

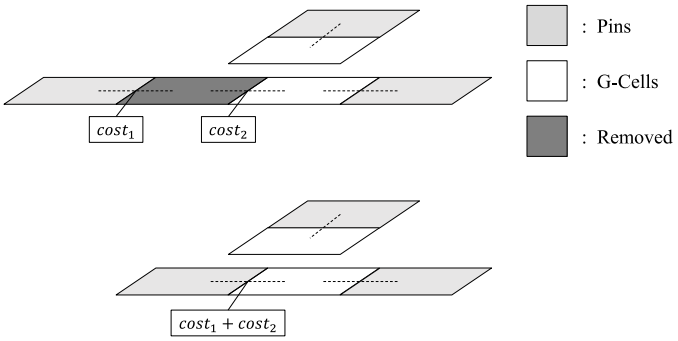


Fig. 8. Routing space compression.

dummy nodes will be inserted to fill up this row, and a new row is added. Note that an infinite cost is applied between different routing tracks placed in the same row, because they are not connected physically.

V. Technical Details

A. 3-D Maze Routing

3-D maze routing is adopted in CUGR because better routing resource allocation can be achieved. A 3-D grid graph consists of several layers of 2-D grid graphs, each of which has one routing direction. Different layers are connected by vias. The sweeping scheme for 3-D routing is alternating between via sweeps (interlayer) and wire sweeps (intralayer).

B. Routing Space Compression

The route guides generated by coarse-grained routing can be further compressed for faster fine-grained routing. An example of routing space compression is shown in Fig. 8. The blackened G-cell can be removed because if it is used, there is only one way to pass through it, and we can just accumulate its cost to a connecting edge.

Routing region compression is also helpful in shortening the routing track length to be smaller than or equal to 1024 in some large benchmarks, which eliminates the need to design new kernels as discussed in Section IV-B.

C. Kernel Call Reduction

A typical routing flow with GAMER is shown in Fig. 10. Connecting one pin in a net requires $(2 \times \text{Iter} + 1)$ kernel calls, where Iter is a parameter controlling the number of sweep alternations. Therefore, routing a net with p pins needs $(p - 1) \cdot (2 \cdot \text{Iter} + 1)$ kernel calls in total.

Too many kernel calls will bring a larger overhead due to kernel launching and memory loading between GPU global memory and kernel shared memory. In practice, the route guides of most nets only span locally and are small enough for all necessary information to be kept by the shared memory of a block. These sufficiently small nets can be more efficiently routed with one block of threads (e.g., $\langle\langle\langle 11\,024\rangle\rangle\rangle$) handling the combined work of $(p - 1) \cdot (2 \cdot \text{Iter} + 1)$ kernels.

This technique is impacted by the size of the routing region, so it is affected by the choice of C in Algorithm 4. To apply kernel call reduction when the fine-grained routing region has

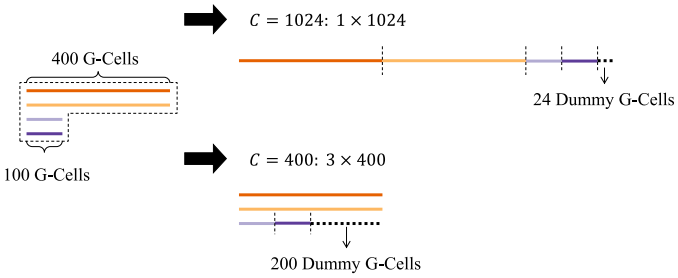


Fig. 9. Routing track extraction and placement results with different column sizes.

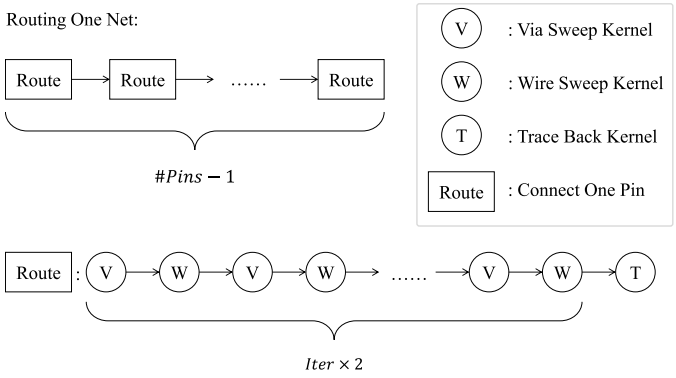


Fig. 10. Many kernels in the routing flow.

no more than 1024 G-cells, C should be set to 1024 for better performance. An example is shown in Fig. 9. The routing tracks are extracted and placed on a rectangle with 1024 (400) G-cells per row. The resulting routing region size is 1×1024 (3×400). Kernel call reduction cannot be applied when C is 400, because 3×400 is larger than 1024, the threshold of applying it. Setting C to 1024 can guarantee the kernel call reduction technique to be applicable if the total length of the routing tracks is within 1024.

D. Net-Level Parallelism

Net-level parallelism is widely used in global and detailed routing. The parallelism comes from routing a batch of nets whose bounding boxes do not overlap simultaneously. There are two common ways to deal with concurrent routing. One way is to route a batch of nets within their own bounding boxes, so that their routings do not affect each other. For the other approach, the routing region is not restricted and each net is routed on a local copy of the grid graph. After routing a batch of nets, all the results are collected and applied in the global grid graph. CUGR adopts the second approach of batch routing. GAMER also implements net-level parallelism by using CUDA streams. A CUDA stream is a kernel queue in which kernels are executed sequentially. Kernels from different streams can be executed concurrently. Therefore, one CUDA stream is used for one net in a batch, and all the kernels for routing a net are inserted into the corresponding streams.

E. Numbers of Alternating Sweeps

GAMER searches for the shortest path by alternating sweeps. The number of sweep alternations (Iter in Fig. 10)

TABLE I
SEQUENTIAL MAZE ROUTING AND GAMER

Grid Graph Size	#Pins	Running Time (s)		Speedup
		GAMER	Maze Routing	
128×128	4	0.04	0.29	7.52×
128×128	8	0.08	0.62	7.25×
128×128	16	0.08	0.63	7.59×
256×256	4	0.07	1.29	17.36×
256×256	8	0.14	2.51	17.71×
256×256	16	0.29	4.93	17.25×
512×512	4	0.16	5.57	33.92×
512×512	8	0.25	11.44	45.14×
512×512	16	0.40	24.60	61.53×
1024×1024	4	0.34	30.40	90.02×
1024×1024	8	0.65	74.51	115.05×
1024×1024	16	1.25	152.52	121.78×

is related to the number of vias in an optimal route. In the coarse-grained routing of GAMER, it is set to a small number 5, because the coarse-grained graph is small and the coarse-grained routing aims at providing a highly routable region based on some rough estimation. Sometimes, GAMER is not able to find an optimal route in terms of cost with five alternations, but the route may actually be good because it has less via usage. The alternation number of the fine-grained routing should be larger because it is applied in a finer grid graph, and it is set to 8.

VI. EXPERIMENTAL RESULTS

A. Environment

Our experiments are conducted on a 64-bit Linux workstation with Intel Xeon Silver 4114 CPUs (2.20 GHz, 40 cores) and 256-GB memory. One NVIDIA GeForce RTX 3090 graphics card is used.

B. 2-D Routing

A simple 2-D maze routing experiment is conducted and the results are shown in Table I. In this experiment, we randomly generate nets with different pin numbers on grid graphs of different sizes to have a simple and direct comparison traditional sequential maze routing and GAMER. Dijkstra's algorithm (A* search that does not include the estimated distance to the destination as the cost) is used in the traditional maze routing. The congestion costs of the grid graphs are generated as follows. 91% of the wire costs are randomly selected from the set $\{1, 2, 3\}$. The remaining 9% is evenly divided into 9 parts, and the ith part has a wire cost of $3 + 2^i$ where $i = 1, \ldots, 9$. This is to simulate a common situation that most of the routing region is uncongested while a small portion is highly congested. A directional change will have a cost of 50, because a directional change indicates via usage, which is usually expensive. For each case, five different grid graphs are randomly generated according to the aforementioned scheme and each grid graph has five different random nets. In this experiment, GAMER will perform 11 iterations of vertical and horizontal sweeps, and it produces the same results as the sequential maze router. As shown in Table I, GAMER achieves up to 121× speedup over the traditional sequential maze router.

TABLE II
BENCHMARK DETAILS

Benchmark	Grid Graph Size		#Layers	#Nets	#Pins
	Original	Coarse-Grained			
ispd18_test5	619 × 613	124 × 123	9	72394	318195
ispd18_test8	905 × 883	181 × 177	9	179863	793289
ispd18_test10	606 × 522	122 × 105	9	182000	811761
ispd19_test7	1053 × 1011	211 × 203	9	358720	1584844
ispd19_test8	1202 × 1138	241 × 228	9	537577	2376399
ispd19_test9	1337 × 1433	268 × 287	9	895252	3957481
ispd18_test5_metal5	619 × 613	124 × 123	5	72394	318195
ispd18_test8_metal5	905 × 883	181 × 177	5	179863	793289
ispd18_test10_metal5	606 × 522	122 × 105	5	182000	811761
ispd19_test7_metal5	1053 × 1011	211 × 203	5	358720	1584844
ispd19_test8_metal5	1202 × 1138	241 × 228	5	537577	2376399
ispd19_test9_metal5	1337 × 1433	268 × 287	5	895252	3957481

C. Integration Into CUGR

GAMER is integrated into CUGR to demonstrate the effectiveness of GAMER in modern global routers. Both the coarse-grained maze routing and the fine-grained maze routing in CUGR are replaced by GAMER. Other parts remain unchanged. Experiments are conducted using the benchmark suite from the 2019 CAD contest at ICCAD on global routing [14], and the same evaluation scheme is adopted. The benchmark details are shown in Table II. The route guides generated by a global router is fed to an academic detailed router Dr.CU 2.0 [15] and detailed routing is conducted. The detailed routing results provided by Dr.CU 2.0 are evaluated by Innovus v18.12-s106_1. The comprehensive evaluation considers wire length, via usage, nonpreferred usage (out-of-guide, off-track and wrong-way) and various design rule violations (shorts, min-area and spacing). Smaller scores are better. The results are shown in Table III and Fig. 12. The benchmarks ending with *metal5* have five metal layers while others have nine metal layers. Eight threads are used by CUGR. Applying GAMER in CUGR can bring an average of 19.85× and 2.59× speedup in the coarse-grained maze routing and the fine-grained maze routing, respectively. An overall 2.7× speedup is achieved without any quality degradation. The maximum GPU memory consumption is 7460 MiB, which occurs in benchmark ispd19_test9_metal5.

In fact, the quality is improved by less than 1%. This is probably because GAMER minimizes via usage by restricting the number of alternating sweeps. The decomposition of the quality scores of GAMER and CUGR is shown in Table IV. GAMER has better quality of results in wire length, via usage, nonpreferred usage and design rule violations.

To see why the large speedup of maze routing becomes less significant in the whole global routing process, benchmark ispd19_test9 is studied and the running time breakdown is shown in Fig. 11. "Others" in the figures includes input, parsing, preprocessing, and output. In CUGR, maze routing is

TABLE III
RUNNING TIME COMPARISON BETWEEN CUGR AND GAMER

Benchmark	Coarse-Grained Time (s)			Fine-Grained Time (s)			Total Time (s)			Quality Score	
	CUGR	GAMER	Speedup	CUGR	GAMER	Speedup	CUGR	GAMER	Speedup	CUGR	GAMER
ispd18_test5	29.15	1.08	26.98	1.91	1.17	1.63	73.08	47.70	1.53	16104127	16146848
ispd18_test8	185.21	6.83	27.13	8.22	5.58	1.47	282.90	99.72	2.84	37937622	37982628
ispd18_test10	257.22	8.98	28.66	18.41	5.17	3.56	373.25	118.59	3.15	40593601	40942190
ispd19_test7	488.70	11.45	42.67	11.63	7.84	1.48	652.42	170.61	3.82	88481579	86843860
ispd19_test8	207.28	9.22	22.48	7.07	9.02	0.78	431.06	226.44	1.90	128287651	127394338
ispd19_test9	308.81	11.72	26.34	7.29	7.09	1.03	620.88	326.84	1.90	201500802	199734785
ispd18_test5_metal5	41.96	7.49	5.60	20.36	6.70	3.04	93.48	40.49	2.31	16206355	16258160
ispd18_test8_metal5	147.38	22.09	6.67	60.89	14.61	4.17	289.95	100.72	2.88	37313105	37334467
ispd18_test10_metal5	226.19	27.18	8.32	76.63	15.68	4.89	399.13	114.33	3.49	46068410	45991227
ispd19_test7_metal5	297.36	15.72	18.92	30.29	12.89	2.35	434.59	136.02	3.20	82368279	81140446
ispd19_test8_metal5	442.85	29.04	15.25	61.31	22.34	2.74	670.22	224.17	2.99	126219722	126706049
ispd19_test9_metal5	427.64	46.91	9.12	72.96	18.48	3.95	732.86	304.05	2.41	197686583	196779131
9-Layer Average			**29.04**×			**1.66**×			**2.52**×		
5-Layer Average			**10.65**×			**3.52**×			**2.88**×		
Average			**19.85**×			**2.59**×			**2.70**×	84897320	**84437844**

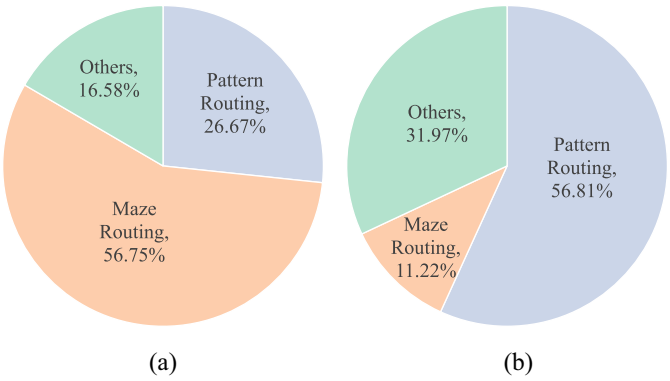


Fig. 11. Running time breakdown of ispd19_test9. (a) CUGR. (b) CUGR+GAMER.

the most time-consuming part. When GAMER is applied, the percentage of maze routing drops from 56.75% to 11.22%, and pattern routing becomes the most time-consuming step.

D. Discussions on Different Speedup Results

Tables I and III seem to show that the 2-D version has a larger speedup over the 3-D version. In fact, the average grid graph size of the 3-D maze routing is 191 × 187 with nine or five metal layers, which can be approximated as a 505 × 495 2-D grid graph. Therefore, the speedup (19.85×) of the 3-D results should be compared with the speedup (46.9×, the average of 33.92, 45.12, and 61.53) on 2-D grid graphs of size 512 × 512. The 2.36× gap between the speedups of 2-D and 3-D may be due to the use of single thread in 2-D routing and multiple threads in 3-D routing. Using one thread is slower than using multiple threads, so the single-threaded version can be accelerated by GPU more significantly.

Table III shows that 9-layer designs have more significant speedup (29.04×) in the coarse-grained routing than 5-layer designs (10.65×). This is because 9-layer designs have: 1) larger grid graphs and 2) less utilization of the eight CPU threads by CUGR. Here is an example to better illustrate the second reason. On average there are 2.48 and 14.43 nets being routed simultaneously by CUGR on ispd18_test10 and ispd18_test10_metal5, respectively. Although 8 threads are provided, only 2.48 threads on average will be active when CUGR is running ispd18_test10. This leaves more room for acceleration using GAMER, which does not rely on this net-level parallelism as much as CUGR. A comparison of the coarse-grained routing time between the single-threaded (light blue bars) and the multithreaded (dark blue bars) CUGR is shown in Fig. 13. It is clearly seen that multithreading brings more speedup to designs with five metal layers. However, it is the opposite when it comes to the fine-grained maze routing. The 9-layer and 5-layer designs are accelerated by 1.66× and 3.52×, respectively. A possible reason is that the fine-grained routing of CUGR on 9-layer benchmarks is so fast (9 s on average) that the overhead of applying GAMER becomes much more significant.

E. Experiments on Alternation Numbers

The number of alternations limits the via usage in a route. Small numbers may result in quality degradation, while large numbers could increase the running time.

The quality scores of different alternation numbers are shown in Table V. The scores are similar when the number of alternations is greater than or equal to 5. However, conducting only three alternations lead to 2% quality degradation.

We further study the effect of different numbers of alternating sweeps on via usage, and the experimental results are shown in Table VI. Intuitively, larger alternation numbers allow more vias to be used and, therefore, should result in larger via numbers. However, the results show that this is not the case. The reasons could be as follows.

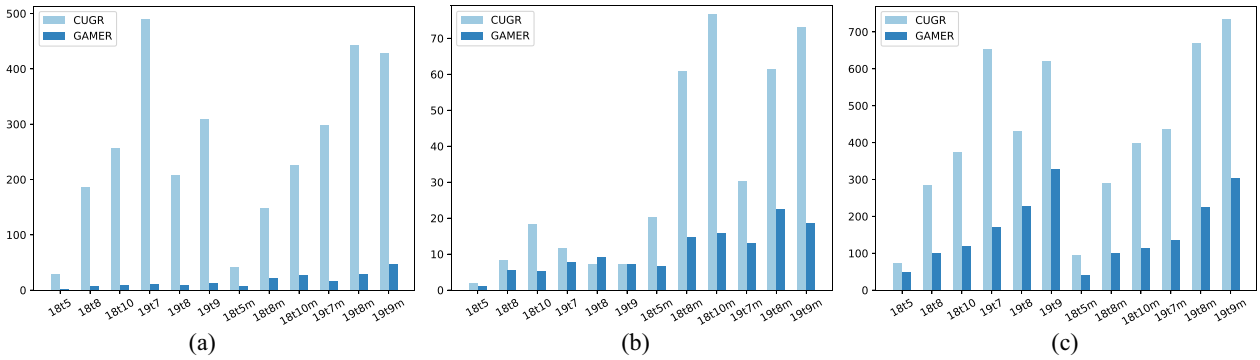


Fig. 12. Running time comparison between CUGR and GAMER. (a) Coarse-grained maze routing. (b) Fine-grained maze routing. (c) Complete global routing.

TABLE IV
DECOMPOSITION OF QUALITY SCORES

Benchmark	Wires		Vias		Non-Preferred Usage		Design Rule Violations	
	CUGR	GAMER	CUGR	GAMER	CUGR	GAMER	CUGR	GAMER
ispd18_test5	13755664	13765367	1854248	1853140	159197	169447	335018	344545
ispd18_test8	32751714	32787737	4692100	4697718	264427	276921	229381	218682
ispd18_test10	34054968	34122192	4991064	4981788	885532	1011274	662038	1042655
ispd19_test7	61002414	60965717	16276308	16188764	1423304	1005572	9779553	8569500
ispd19_test8	93420226	93355897	25802196	25368768	1335239	1042807	7729990	7624603
ispd19_test9	141254193	141164233	42973076	42253008	2181815	1665822	15091718	14610771
ispd18_test5_metal5	13984870	13951546	1830804	1849656	129581	156142	261100	293815
ispd18_test8_metal5	32253644	32270995	4493288	4541220	344337	456485	221836	275589
ispd18_test10_metal5	35389693	35661566	4871744	4851486	1439283	1567962	4367690	1847210
ispd19_test7_metal5	54592500	54542449	16240576	16137624	1531423	1116679	10003780	9690458
ispd19_test8_metal5	90539408	90648239	25557880	25246580	1473764	1418067	8648670	9683245
ispd19_test9_metal5	136688095	136555120	42557508	41993412	2341136	1907321	16099845	16826763
Average	**1.000**	**1.000**	1.011	**1.000**	1.145	**1.000**	1.034	**1.000**

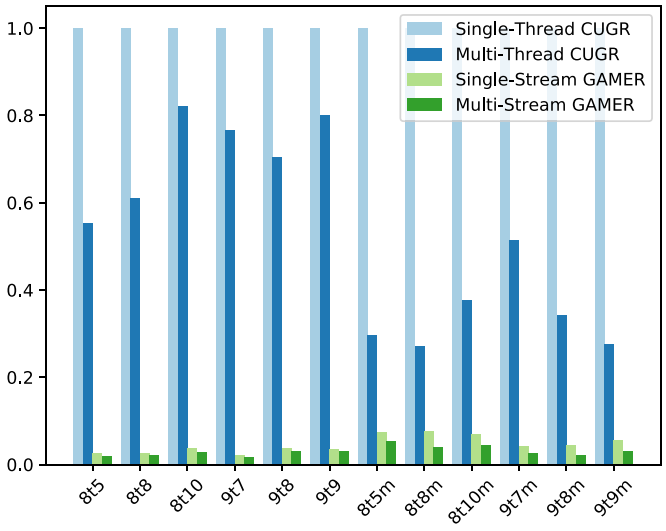


Fig. 13. Normalized coarse-grained maze routing time with/without net-level parallelism.

1) More alternations will result in more vias only when using more vias gives a shorter path to connect two pins. If the via cost in the cost scheme is large, the shortest path may often be using only a few vias. In this case, a small alternation number suffices and any extra alternations are useless.
2) In a design of nine layers, 1–8 vias can be used in a via sweep, because a path can go from layer 1 to layer 2 or from layer 1 to layer 9 in a via sweep. This uncertainty makes it hard to measure via usage simply by the number of alternations.

F. Net-Level Parallelism

Net-level parallelism is implemented by multithreading in CUGR and multiple streams in GAMER. Experiments of GAMER and CUGR with and without net-level parallelism are shown in Fig. 13. The benchmarks from left to right in Fig. 13 are the same as the benchmarks from top to bottom in Table II. Eight threads are used by the multithread version of CUGR. The stream number is not limited in the multi-stream GAMER, but CUDA supports concurrent execution of up to 128 kernels. Fig. 13 shows that GAMER has significant speedup even when the nets are routed sequentially (single-stream). This is because the speedup of GAMER

TABLE V
QUALITY SCORES OF DIFFERENT ALTERNATION NUMBERS IN COARSE-GRAINED MAZE ROUTING

Benchmark	#Alternations†					
	3	5	7	9	11	13
ispd18_test5	16100010.90	16146847.96	16125797.79	16080807.71	16080807.71	16080807.71
ispd18_test8	37921673.82	37982628.19	37943090.41	37939064.41	37956210.00	37945448.01
ispd18_test10	41106098.99	40942189.62	40888701.69	40908280.17	40972284.86	40933798.40
ispd19_test7	86918165.03	86843860.49	86843894.12	86761094.23	86558666.09	86519723.44
ispd19_test8	127343982.67	127394338.04	127359098.88	127477176.33	127335671.05	127253942.12
ispd19_test9	200174314.72	199734785.30	199483537.53	199365558.02	199581634.18	199392863.48
ispd18_test5_metal5	18033299.14	16258159.68	16180418.73	16174145.04	16168705.10	16174898.74
ispd18_test8_metal5	37251862.14	37334466.92	37194340.58	37163973.29	37182145.07	37182145.07
ispd18_test10_metal5	46129136.09	45991227.08	45691905.92	45656882.58	45503747.04	45503747.04
ispd19_test7_metal5	81413075.77	81140446.19	80927450.87	80962605.23	81244336.06	81244336.06
ispd19_test8_metal5	136072357.69	126706049.45	126506610.85	126624342.48	126619343.79	126497021.12
ispd19_test9_metal5	207978404.14	196779131.10	196243301.86	196232895.57	196306504.62	196335997.81
Average	86370198.43	84437844.17	84282345.77	84278902.09	84292504.63	84255394.08

†An alternation consists of a via sweep followed by a wire sweep.

TABLE VI
VIA NUMBERS OF DIFFERENT NUMBERS OF ALTERNATING SWEEPS IN COARSE-GRAINED MAZE ROUTING

Benchmark	#Alternations					
	3	5	7	9	11	13
ispd18_test5	926030	926603	925525	926348	926348	926348
ispd18_test8	2347936	2348715	2348270	2348217	2349291	2348410
ispd18_test10	2489406	2489010	2489595	2489419	2489214	2488920
ispd19_test7	4043791	4044789	4045696	4045134	4044852	4044001
ispd19_test8	6344577	6343340	6343565	6342350	6344226	6343807
ispd19_test9	10561269	10562240	10563980	10563793	10562874	10564231
ispd18_test5_metal5	922836	924290	923450	924052	923734	923876
ispd18_test8_metal5	2265280	2265758	2266084	2266718	2266498	2266498
ispd18_test10_metal5	2425927	2427548	2424150	2425333	2425752	2425752
ispd19_test7_metal5	4029575	4032230	4027348	4029839	4028233	4028233
ispd19_test8_metal5	6318565	6307697	6303303	6304993	6306730	6305712
ispd19_test9_metal5	10512942	10495925	10491527	10492096	10493279	10491228
Average	**4432345**	**4430679**	**4429374**	**4429858**	**4430086**	**4429751**

is mostly from pathfinding-level parallelism, which is usually more consistent than net-level parallelism. In fact, the pathfinding-level parallelism of GAMER can already use up all cores of modern GPUs. For example, GeForce RTX 3090 has 10 496 cores, but a wire sweep of benchmark *ispd*19_*test*9 needs $268 \times 287 \times 9 = 692\,244$ threads.

VII. CONCLUSION

This article introduced a novel parallel algorithm for VLSI maze routing. An operation called *sweep* was first introduced and two $O(\log_2 n)$ parallel algorithms for sweeps on an $n \times n$ grid graph were proposed. Maze routing can be achieved by alternating horizontal and vertical sweeps and, thus, can be accelerated by the parallel sweep algorithms. An efficient and useful method of handling non-rectangular routing regions was also designed. By integrating GAMER to the state-of-the-art LEF/DEF-based global router CUGR, experiments showed that the coarse-grained and fine-grained routing in CUGR can be accelerated by 19.85× and 2.59×, respectively, which together contribute to an overall 2.7× speedup of CUGR without any quality degradation.

REFERENCES

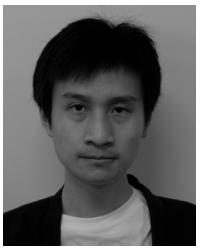
[1] K.-R. Dai, C.-H. Lu, and Y.-L. Li, "GRPlacer: Improving routability and wire-length of global routing with circuit replacement," in *Int. Conf. Comput.-Aided Des. Dig. Tech.*, 2009, pp. 351–356.

[2] M.-C. Kim, J. Hu, D.-J. Lee, and I. L. Markov, "A SimPLR method for routability-driven placement," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2011, pp. 67–73.

[3] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Amsterdam, The Netherlands: Elsevier, 2008, pp. 365–381.
[4] M. Pan and C. Chu, "FastRoute 2.0: A high-quality and efficient global router," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2007, pp. 250–255.
[5] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "Multi-threaded collision-aware global routing with bounded-length maze routing," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 200–205.
[6] C. Chiang, J. Kawa, and Y. Wen, "Routing in multithread environment," in *Proc. 5th Int. Conf. ASIC*, vol. 1, 2003, pp. 203–207.
[7] Y. Shintani, M. Inagi, S. Nagayama, and S. Wakabayashi, "A multithreaded parallel global routing method with overlapped routing regions," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2013, pp. 591–597.
[8] J. He, M. Burtscher, R. Manohar, and K. Pingali, "SPRoute: A scalable parallel negotiation-based global router," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2019, pp. 1–8.
[9] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 5, pp. 709–722, May 2013.
[10] Y. Han, K. Chakraborty, and S. Roy, "A global router on GPU architecture," in *Proc. IEEE 31st Int. Conf. Comput. Des. (ICCD)*, 2013, pp. 78–84.
[11] J. Liu, C.-W. Pui, F. Wang, and E. F. Y. Young, "CUGR: Detailed-Routability-driven 3D global routing with probabilistic resource model," in *Proc. 57th ACM/IEEE Des. Autom. Conf. (DAC)*, 2020, pp. 1–6.
[12] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.
[13] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980. [Online]. Available: https://doi.org/10.1145/322217.322232
[14] S. Dolgov, A. Volkov, L. Wang, and B. Xu, "2019 CAD contest: LEF/DEF based global routing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2019, pp. 1–4.
[15] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Y. Young, "Dr. CU 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2019, pp. 1–7.

Shiju Lin received the B.Eng. degree in computer science and technology from the South China University of Technology, Guangzhou, Guangdong, China, in 2020. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His current research interests include global routing and GPU acceleration in EDA.



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

His current research interests include electronic design automation, combinatorial optimization, and machine learning.

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

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

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

Martin D. F. Wong (Fellow, IEEE) received the B.Sc. degree in mathematics from the University of Toronto, Toronto, ON, Canada, in 1979, and the M.S. degree in mathematics and the Ph.D. degree in computer science (CS) from the University of Illinois at Urbana-Champaign (UIUC), Champaign, IL, USA, in 1981 and 1987, respectively.

He was a Faculty Member with The University of Texas at Austin (UT-Austin), Austin, TX, USA, from 1987 to 2002 and UIUC from 2002 to 2018. He was a Bruton Centennial Professor of CS with UT-Austin and an Edward C. Jordan Professor of ECE with UIUC. From August 2012 to December 2018, he was the Executive Associate Dean of the College of Engineering, UIUC. Since January 2019, he has been with The Chinese University of Hong Kong, Hong Kong, as the Dean of Engineering and the Choh-Ming Li Professor of Computer Science and Engineering. He has published around 500 papers and graduated over 50 Ph.D. students in electronic design automation (EDA). His main research interest is in EDA.

Prof. Wong is a Fellow of ACM.