AceRoute: Adaptive Compute-Efficient FPGA Routing with Pluggable Intra-Connection Bidirectional Exploration

Xinming Wei[1]✉, Ziyun Zhang[1], Sunan Zou[1], Kaiwen Sun[2], Jiahao Zhang[3], Jiaxi Zhang[1]✉, Ping Fan[2], and Guojie Luo[1,4]✉
[1]School of Computer Science, Peking University, Beijing, China
[2]DeePoly Technology Inc., Beijing, China
[3]Undergraduate School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[4]Center for Energy-efficient Computing and Applications, Peking University, Beijing, China
{weixinming,zhangjiaxi,gluo}@pku.edu.cn

ABSTRACT

This paper introduces AceRoute, an adaptive compute-efficient FPGA router that tackles the long-standing issue of lengthy FPGA compilation times given complicated FPGA architectures and designs to synthesize. We thoroughly profile modern FPGA routing patterns and identify the runtime hotspot: routing bottleneck connections in congested designs. However, previous works on routing acceleration hardly target mitigating connection-wise routing difficulties by characterizing device resource expansions and shifting path-exploration modes of connections.

In this work, we propose a bidirectional intra-connection routing paradigm for the first time, which efficiently navigates congested device regions by initiating searches from both source and sink nodes. This approach significantly outperforms traditional unidirectional exploration in congested conditions. Furthermore, for each connection to route, we develop an adaptive strategy to select the optimal search mode online between uni- and bi-directional based on the current congestion situation. Our approach is pluggable and versatile, allowing seamless integration into existing FPGA routers and providing instant speed improvements with merely several hundred lines of code.

Evaluation on FPGA24 contest benchmarks shows that our router, powered by adaptive bidirectional search, achieves over 2.4× and 3.2× faster routing on average with a serial and a 2-thread parallel version, respectively, compared with established routing tools RWRoute and Vivado. Additionally, when integrated into typical partition-based inter-connection parallel routers, our approach overcomes their inherent load-balance problems and amplifies the speedup to RWRoute from 2.2× to up to 5×.

ACM Reference Format:
Xinming Wei, Ziyun Zhang, Sunan Zou, Kaiwen Sun, Jiahao Zhang, Jiaxi Zhang, Ping Fan, and Guojie Luo. 2024. AceRoute: Adaptive Compute-Efficient FPGA Routing with Pluggable Intra-Connection Bidirectional Exploration. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676709

1 INTRODUCTION

The lengthy compilation times have long been an Achilles heel for field programmable gate array (FPGA) technology, and this issue is further exacerbated by the growing complexity of both the modern FPGA architectures and the user designs. As a pivotal backend step in the FPGA compilation flow, routing constitutes a significant portion of the runtime [3], which assigns logic nets to physical routing resources on an FPGA. Given the trend of escalating scale of FPGA devices and applications, there exists an urgent need to speed up the compute-intensive routing process.

Mainstream Routing Paradigm. Before going deeper into the prior art of routing acceleration, it is necessary to review the landscape of the mainstream routing frameworks and analyze their runtime bottlenecks. The principles of *negotiated congestion* [11] and *connection-based* [22, 23, 30] routing now underline most modern FPGA routers [9] and we will elaborate them in § 2.2. For now, it is sufficient to learn that signals are routed iteratively in the unit of *connection*, which identifies a pair of a source and a sink node. The goal is to find node-disjoint paths that complete all connections over a resource graph. Once some paths overlap on a certain resource node, which creates *congestion*, the cost of this node is increased to discourage other connections from routing through it. In each routing iteration, the overlapping connections are ripped up and rerouted to expect that all the conflicts are eliminated.

Runtime Bottleneck. The runtime of an FPGA router is essentially determined by the number of resource expansions when exploring the best-cost path for each connection. The routing of conflicting connections located in highly congested regions substantially increases the resource expansions, and thus dominates the runtime. To route a connection, an A* search beginning from the source explores downhill resource nodes recursively until finding the sink. The exploration prioritizes lower-cost nodes to improve the optimality of the search path. When rerouting connections within highly congested regions, the amount of visited nodes grows exponentially for the purpose of circumventing overused nodes and seeking a viable path. We will analyze quantitatively these runtime bottlenecks in § 4. However, previous works on FPGA routing acceleration hardly identify such a bottleneck and propose corresponding optimizations.

Previous Routing Acceleration Approaches. Most published attempts to speed up FPGA routing focus on parallelism, while others target serial algorithmic enhancements on the negotiated congestion routing paradigm. We categorize these efforts into 1) ***inter-connection***[1], and 2) ***intra-connection*** acceleration by the routing granularity. Inter-connection routers [4, 6, 16, 24, 31] provide connection-level parallelism, usually distributing the routing task by partitioning the connections into *conflict-free* subsets; these partitions are then routed independently and simultaneously. While inter-connection routers can somewhat speed up the routing process, the frequent congested situations prevent the common workload-partitioning methods from being effective [17]. Such problems with load-balancing lead to limited scalability and massive consumption of computing resources (threads or processor cores) for a plain speedup. Unlike inter-connection acceleration, intra-connection does not alter the processing order of connections. It augments the path seeking between source and sink for each connection. Existing intra-connection parallelism [12, 25] approaches are much fewer than inter-connection ones and are not widely used, due to high parallelism overhead and poor determinism [20]. Other serial efforts are devoted to adjusting the cost settings [28, 31], or renewing FPGA architectures [17, 21], which lack generality in terms of actual designs or FPGA devices.

Motivation for Adaptive Bidirectional Exploration. The above previous works rarely take connection-wise routing complexity into account. To address the runtime hotspots when routing connections in highly congested device regions, we propose an adaptive bidirectional exploration (BE) approach for intra-connection routing. Typical A* search, which is unidirectional from source to sink, has dominated modern FPGA routers despite that incremental optimizations of cost settings or parallelism are witnessed. However, unidirectional exploration (UE) is not ideal when dealing with congestion, because it tends to expand excessive resource nodes when encountering a vast amount of overused nodes. BE has been proven effective on other tasks such as route planning for robots or vehicles [26, 27]. In this work, we develop BE intra-connection routing that drives two search frontiers from both the source and the sink with dedicated dual cost settings and pruning strategies. BE greatly outperforms UE in congested conditions, since two frontiers meeting halfway avoids lavish explorations of tempting but sub-optimal paths. Nevertheless, UE behaves better at uncongested cases, where the searched path is somewhat "straightforward" and UE stops right after the search frontier arrives at the sink, while BE needs to weigh in many possible source-sink paths. We will illustrate in detail the above observations in § 5.3. Accordingly, we design an efficient connection-wise adaptive strategy to determine whether to use BE or UE to route the current connection based on historical search experiences. Note that our adaptive BE routing paradigm is ***pluggable***, *i.e.*, programming and integrating it into commercial or academic FPGA routers is seamless and effortless, but the speedup is instant and surprising. When incorporated into inter-connection parallel routers, adaptive BE overcomes their inherent workload-balance problems and provides multiplied speedup, while being orthogonal to their specific parallelism mechanisms.

Our Contributions. We summarize our contributions as below:

[1]Some works are net-based instead of connection-based, performing intra-net or inter-net acceleration, where each net defines one source and one/multiple sink(s) (§ 2.2). For simplicity, we do not differentiate the usage of these two concepts in this section.

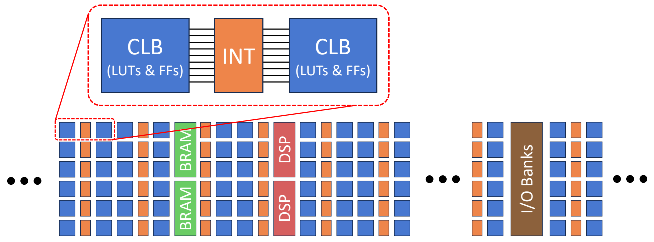

Figure 1: UltraScale+ architecture with columnar resources.

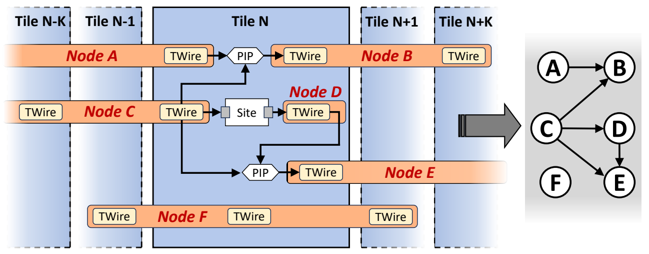

Figure 2: Modeling an example FPGA routing circuitry as a routing resource graph (RRG).

- We profile in depth the runtime of the modern FPGA routing paradigm and discover that the bottleneck connections in congested designs cause routing hotspots (§ 4).
- For the first time, we introduce the notion of bidirectional intra-connection routing (§ 5.1) with lightweight serial and parallel implementations (§ 5.2), and analyze its effectiveness with dedicated case studies and statistical data (§ 5.3).
- We propose an adaptive switching strategy between BE and UE for maximized performance under varied congestion situations (§ 6.1), and enable the seamless integration of adaptive BE into inter-connection parallelism routers (§ 6.2).
- On the FPGA24 contest benchmark, AceRoute powered by serial adaptive BE and 2-thread parallel BE achieve over 2.4× and 3.2× average speedup, respectively, compared with RWRoute and Vivado, the open-source and industrial state of the art routers (§ 7.1). AceRoute also amplifies the speedup (to RWRoute) of a partition-based inter-connection parallel router from 2.2× to 4.4× and 5.0× with serial adaptive BE and 2-thread parallel BE, respectively (§ 7.2). The wirelength overheads incurred by these acceleration techniques are minor.

2 PRELIMINARIES

2.1 FPGA Architecture and Representation

UltraScale+ Architecture. In this study we target AMD/Xilinx Virtex UltraScale+ FPGAs, which feature a column-and-grid layout. Figure 1 depicts the basic architectural elements. At an abstract level, a device is created by assembling a grid of *tiles* of different types, and tiles in the same column share a type. Each tile type defines unique *tile wires (TWires)* totally contained within this tile, and *programmable interconnect points (PIPs)* that provide configurable connections between tile wires. A tile type also defines *sites*, and a site includes a group of *basic elements of logic (BELs)* and their connectivity. A site can be connected to tile wires through *site pins*. Common tile types include *configurable logic block (CLB)*, block RAM (BRAM), DSP, I/O bank, and *interconnect (INT)*. INT tiles are switch box tiles where most PIPs are present. The segmented interconnect network spanning one or multiple INT tiles is designed for the optimal signal transportation among fabric resources such as CLBs, DSPs, and BRAMs.

Routing Resource Graph and Task Formulation. Using routing resource graph (RRG) to represent FPGA device resources is a common practice in FPGA routing. Figure 2 illustrates the mapping of routing resources to an RRG. As the fundamental components in an RRG, a *node* is a collection of electrically connected tile wires (TWires) across one or more tiles, and an *edge* denotes a PIP or a site. UltraScale+ FPGAs supply considerable routing resources. For instance, the RRG of device xcvu3p consists of over 28 million nodes and 125 million edges. The extensive scale of RRG introduces unprecedented complexity to routing.

The routing task can be formulated as finding a routing tree of RRG nodes for each *net* in the design such that 1) each tree connects the net source and all the net sinks, and 2) no two trees contain the same node. The optimization target depends on the routing mode. *Wirelength-driven* routing aims to minimize wirelength for a compact and potentially less power-consuming design, while *timing-driven* routing prioritizes meeting timing requirements to avoid signal propagation delays, often at the expense of longer paths or more resources. We focus on wirelength-driven routing in this work, though switching to timing-driven is convenient via adjusting the costs specified in § 2.2.

2.2 Negotiated Congestion Routing

PathFinder-Based Routing. FPGA routers are typically based on the negotiated congestion mechanism introduced by PathFinder [11]. For all nets, the router iteratively tries to find disjoint routing trees in the RRG. Historically, *net-based* rip-up-and-reroute proceeds until no resource nodes are illegally shared in each iteration. Recent high-performance routers are mostly *connection-based* [22, 23, 30, 31], splitting up each net into a set of source-sink connections and routing the connections independently. Connection-based routers are more efficient because only congested connections are rerouted instead of all nets. A resource node can be shared by multiple connections only if these connections belong to the same net or, equivalently, have the same source.

To eliminate congestion and minimize the wirelength, negotiated-congestion-based routers employ the congestion cost function C_{total} of a given node n when routing each connection:

$$C_{total}(n) = C_{prev}(source, n) + C_{node}(n) + C_{exp}(n, sink) \qquad (1)$$

$$\text{where } C_{node}(n) = \frac{b(n) \cdot p(n) \cdot h(n)}{1 + share(n)}$$

The node cost $C_{node}(n)$ involves the base cost $b(n)$ for wirelength, the present and historical congestion cost $p(n)$ and $h(n)$ that penalize the sharing of n by multiple connections, and the sharing factor $share(n)$ for the number of connections in the same net that share this node. The design of $C_{node}(n)$ allows connections to share resources initially and gradually increases the penalty for resource overuse in later iterations. The upstream path cost $C_{prev}(source, n)$ accumulates the node cost of nodes along the search path from the source, while the expected wirelength cost $C_{exp}(n, sink)$ estimates the downstream cost, basically related to the Manhattan distance between n and the sink.

Intra-Connection A* Search Given the above cost settings, each connection is routed with A* search [5]. Compared with breadth-first search (BFS), A* is the so-called best-first search that selects the most promising neighboring node for expansion, typically implemented with a priority queue. Each node u is ranked with an evaluation function[2] $f(u) = g(u) + e(u)$, where $g(u)$ denotes the shortest path found so far from the source to u, and $e(u)$ is the heuristic function for the path length from u to the target. The heuristic function $e(u)$ is said to be *admissible* if it never overestimates the actual cost from u to the target, which guarantees the optimal solution. Typically, an A* search is *unidirectional*, starting from the source and terminating when reaching the target.

3 EXPERIMENTAL SETUP

This section describes the general experimental setup applied to all the experiments in the subsequent sections. As mentioned in § 2.1, we target AMD/Xilinx Virtex UltraScale+ FPGAs in this study.

Environment. We perform all experiments on an Ubuntu 22.04 LTS server with a 48-core (96-thread) Intel Xeon Platinum 8488C CPU and 768GB DDR4 RAM.

Baselines. For serial FPGA router baselines, we select Vivado v2023.1 [2] and RWRoute [30], the commercial and academic representatives compatible with the latest FPGA architectures like UltraScale+. RWRoute is an open-source negotiated-congestion-based router provided by Java-based RapidWright [10].

AceRoute Implementation. We first implement a basic serial version with C++ from scratch, on the foundation of RWRoute. Then we revolutionize it based on our proposed serial and parallel bidirectional enhancements. AceRoute also supports inter-connection parallelism based on net bi-partition [16], a common partition-based scheme. We utilize C++ std::thread and Taskflow [7] for intra- and inter-connection parallelism, respectively. AceRoute contains ~6000 lines of code excluding third parties.

Benchmarks. We use FPGA24 routing contest benchmarks [8] to evaluate all the routers. The benchmark contains a series of pre-placed and partially routed (clock and power nets) physical netlists. The benchmark suite has two parts: public and hidden. Due to the large amount of designs, we utilize the designs in the public part while removing 2 smallest ones from it, and adding the 3 largest ones from the hidden part, for the evaluations in this paper.

Runtime Statistics. We use wall clock time to assess routing performance. Each routing time reported in experiments will not include the time to load device resources or read/write the netlist for better comparison, though AceRoute achieves similar data I/O efficiency with Vivado or RWRoute via plentiful engineering efforts.

4 RUNTIME BREAKDOWN AND PROFILING

In this section, we analyze the runtime properties and bottlenecks of connection-based negotiated congestion routing. Observations and conclusions here set the stage for our next methods. We conduct

[2]In FPGA routing, the total cost $f(u) = C_{total}(u)$, the actual cost $g(u) = C_{prev}(source, u) + C_{node}(u)$, and the estimated cost $e(u) = C_{exp}(u, sink)$.

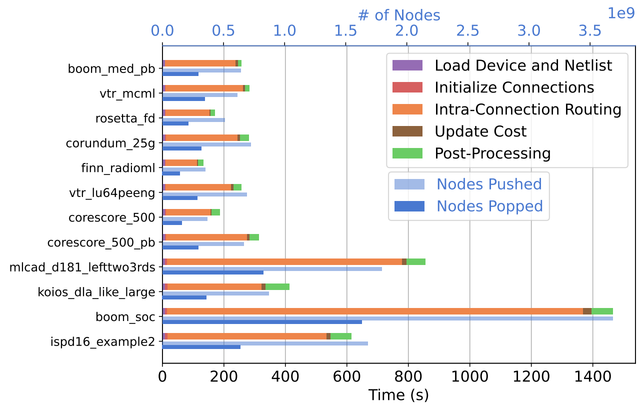

Figure 3: Runtime breakdown of the routing process and the corresponding numbers of heap pushes and pops.

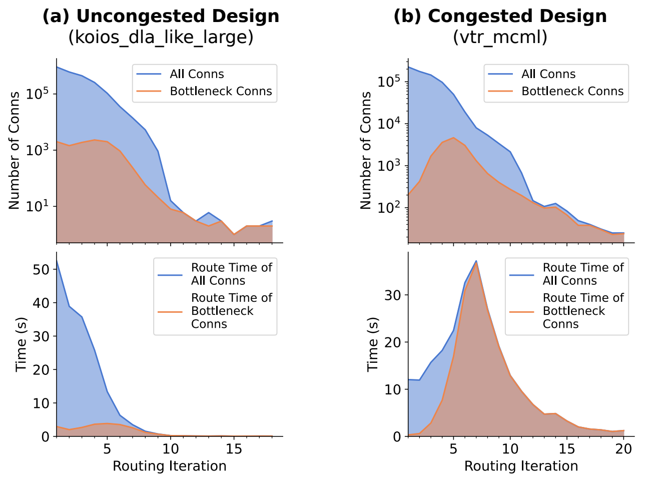

Figure 4: Impact of bottleneck connections (conns) on runtime under different levels of congestion.

the experiments in this section with the basic serial UE version of AceRoute on the FPGA24 public benchmark.

4.1 Runtime Proportion and Performance Indicator of Intra-Connection Routing

To demonstrate the necessity of accelerating intra-connection routing and the correlation between resource expansions and routing runtime, we measure the runtime breakdown of each routing stage and the number of heap operations in the whole routing process.

As shown in Figure 3, for each design, we break down the routing process into 5 stages, including data read/write, initialization of connections, intra-connection routing, cost updating between iterations, and post-processing. Accompanied is the number of nodes pushed or popped to/from the heap, shown in the blue bars with data aligned to the top blue axis. We obtain two conclusions from the results. 1) It is evident that intra-connection routing takes up the majority of routing runtime, with an average 83.7% across all designs, since it is the core of connection-based routing. 2) Heap pushes and pops are strongly correlated to the actual runtime of intra-connection routing. The nature of A*-based algorithms determines that heap operations are good indicators of search efficiency.

4.2 Quantifying Routing Bottlenecks

Since conflicting connections are ripped up and rerouted in each routing iteration, what connections consume most of the runtime? Now we conclude that 1) ***bottleneck connections*** in 2) ***congested designs*** drive the runtime up. For 1), we define bottleneck connections as those that take more than 1*ms* to route in an arbitrary routing iteration. In practice, they are difficult to route with large amounts of resource expansions. The impact of bottleneck connections is much more prominent in 2) congested designs, where higher congestion is more prevalent and each bottleneck connection is harder to handle.

In Figure 4, we demonstrate the effect of bottleneck connections under two different levels of congestion. For both design (a) and (b), bottleneck connections constitute a rather small portion of the total until the very last routing iterations. But when it comes to runtime impact, the distinctions are clear. For (a), the runtime contribution of bottleneck connections remains low until the several unresolved connections in the final routing stage. For (b), bottleneck connections introduce significant runtime overhead as routing proceeds, even if they are the minority of all connections. In the next section, we will introduce our BE intra-connection routing approach to address bottleneck connections and speed up the routing.

5 BIDIRECTIONAL EXPLORATION

In view of the runtime profiling results and analysis in the last section, we now propose bidirectional intra-connection routing. We first introduce the strategic details of the algorithm design in § 5.1, and then describe an efficient implementation of the parallel version in § 5.2. Finally, in § 5.3, we experimentally analyze the performance variance of BE and UE under different levels of congestion.

5.1 Algorithm Design

We develop our BE-based intra-connection routing inspired by previous works on bidirectional A* algorithm [13–15, 19]. BE executes two search processes simultaneously: one *forward* process from the source and the other *backward* process from the sink. BE does not necessarily run in parallel; we present a serial version alternating the execution of each search process, together with a parallel one in § 5.2. In the backward process, a *reverse* RRG is created, where each PIP or site edge is reversed.

Given the symmetry of two search processes, we demonstrate the algorithm with one side and denote every variable on the other with a bar. $\overline{s}$, $\overline{C}$, $\overline{visited}$, and $\overline{parent}$ denote the source, cost, visited flag, and node parent of the opposite search, respectively. The execution of BE stops when one process stops. Algorithm 1 gives the explicit BE algorithm viewed by one process. To restrict hopeless node expansions, BE uses a read/write variable $\mathcal{L}$ shared by both processes, which saves the best path cost found so far. $\mathcal{L}$ is employed in the pruning criteria (Line 13) together with C^*, the lowest total cost of nodes on the search frontier.

Compared with classic bidirectional A*, our BE routing in Algorithm 1 modifies node labeling and heuristic function to balance both searching efficiency and wirelength. In fact, many FPGA routers make similar adjustments to *unidirectional* A*. Such modifications reduce node expansions with compromised optimality. For example, a node can only be visited once in our approach, while in traditional A* it may be expanded multiple times. The expected downstream cost C_{exp} is multiplied by a *directedness* factor that determines how aggressively the router explores toward the sink. Larger directedness improves runtime but may result in suboptimal routes since admissibility may not be satisfied. A suitable directness value will not introduce a noticeable reduction in result quality.

Algorithm 1: BE of One Search Process

```
Input: RRG $G = (V, E)$ with source $s$ and sink $\bar{s}$.
Output: A source-sink path $(s, u_1, \dots, u_n, \bar{s})$.
$\mathcal{L} = \infty$;                                   // The best path cost
$V_{result} = NULL$;                          // Internal node in the result path
foreach $v \in V$ do
    $C_{prev}(s, v) = \infty$;
    $v.visited = false$;
$s.visited = true$;
$C_{prev}(s, s) = 0$;
$C^* = C_{total}(s)$;      // The lowest cost of nodes on search frontier
$Q$ = priority_queue();          // Prioritize lower $C_{total}(v), v \in V$
$Q$.push($s$);
while not $Q$.empty() do
    $u = Q$.pop();
    if $C_{total}(u) \geq \mathcal{L}$ or $C_{prev}(s, u) + \overline{C^*} - \overline{C_{exp}}(u, t) \geq \mathcal{L}$ then
        continue;
    $up\_cost = \min(C_{prev}(s, u), C_{prev}(s, u.parent) + C_{node}(u))$;
    foreach $(u, v) \in E$ do
        if $v.visited == true$ then continue;
        $v.visited = true$;
        $v.parent = u$;
        $C_{prev}(s, v) = up\_cost$;
        $Q$.push($v$);
        if $\overline{v.visited} == true$ then
            if $C_{prev}(s, v) + \overline{C_{prev}}(t, v) + C_{node}(v) < \mathcal{L}$ then
                $\mathcal{L} = C_{prev}(s, v) + \overline{C_{prev}}(\bar{s}, v) + C_{node}(v)$;
                $V_{result} = v$;
    $C^* = C_{total}(Q.\text{top}())$;
Restore and return the best $s$–$t$ path with $V_{result}$, $V_{result}.\overline{parent}$,
  and $V_{result}.\overline{parent}$ recursively;
```

5.2 Parallel Implementation

Our parallel BE is built upon the notion in §5.1 but tailored for multi-threaded execution with shared memory. We parallelize the forward and backward searches instead of interleaving them. The need for mutual exclusion is small since most variables are written only by one search side, lowering the parallelism overhead.

As shown in Figure 5, we use two threads for forward and backward searches respectively to route each connection. Each thread maintains a local priority queue. Two barriers are set to synchronize

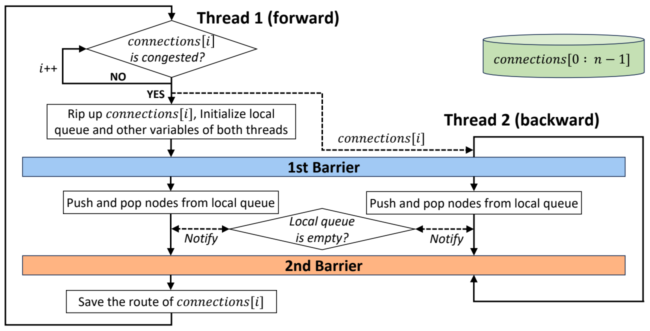
Thread 1 (forward)
$connections[i]$ is congested?
i++
NO
YES
$connections[0 : n-1]$
Rip up $connections[i]$, Initialize local queue and other variables of both threads
Thread 2 (backward)
$connections[i]$
1st Barrier
Push and pop nodes from local queue
Push and pop nodes from local queue
Local queue is empty?
Notify
Notify
2nd Barrier
Save the route of $connections[i]$

Figure 5: Parallel BE routing with 2 threads.

the execution of the two threads: the backward thread waits until the forward thread finishes saving the routing of the last connection and initializing the next one to route, and then both threads route the same connection concurrently until one of them terminates with the result path. In this way, two threads are reused for each connection without the expensive joining and creating. Parallel BE is not deterministic because the scheduling of two search threads is random, and intra-connection routing does not promise optimality.

5.3 Is Bi-Directional Always More Efficient?

The different natures of BE and UE imply that each one of them may outperform the other under some circumstances. Here we inspect the root cause of the distinctions through elaborate experiments. To guarantee the fairness of comparison, we ensure that each connection is routed with either UE or BE given the same routing status. Specifically, we modify the common routing procedure as below:

- In each routing iteration, unrouted or conflicting connections are ripped up and routed one by one in a given order.
- For each connection to route:
  - Route it with BE, record the statistics.
  - Undo the route, *i.e.,* discard the intermediate results and the searched path produced by BE.
  - Route it with UE, record the statistics.

In this way, we restrict UE and BE to route each connection based on the same routing checkpoint. This procedure is the basis of the following two experiments.

Exp I: Case Study of a Single Connection. To visualize the performance variations between UE and BE at different routing stages or degrees of congestion, we compare the resource expansions and runtime of routing a single connection with UE or BE at both an initial and a later routing iteration. As shown in Figure 6, we use connection 32221 routed in iteration 1 and 3 from design `vtr_mcml` for illustration. In this specific case, it happens that UE and BE find the same path in both iteration 1 and 3. We compare the exploration overhead of UE and BE in these two iterations, respectively. 1) In the first routing iteration, both UE and BE visit limited nodes, when the node cost does not penalize a lot on overuses and congestion is rare to be found. However, BE tends to pop more nodes from the heap than UE, as it tries to find multiple plausible paths and choose the best one, while UE stops once reaching the sink. 2) In iteration 3, with the increasing congestion, BE explores much fewer nodes than

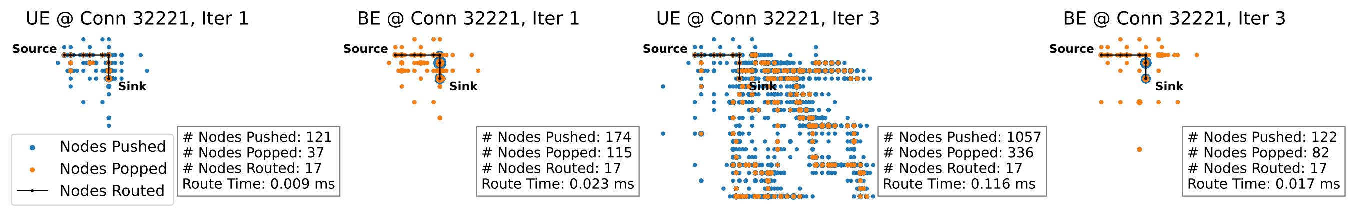

Figure 6: Comparison of routing a specific connection with UE/BE at different routing stages. Each node (wire) is positioned at the tile it ends. The larger dot denotes more nodes expanded at the same tile position.

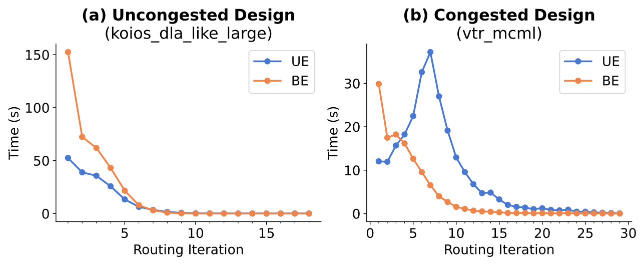

Figure 7: Runtime comparison between UE and BE on the same designs in Figure 4 with different levels of congestion.

UE. UE performs numerous wasteful expansions before reaching the sink, because the high-cost nodes around the sink drive the searching frontier to explore misleading directions. In contrast, BE efficiently seeks the same path as two searching frontiers head towards each other. Here the number of expansions is too small to indicate a bottleneck connection, yet it is suitable for visualization. In many cases, UE can explore thousands of times more nodes than BE in congested situations.

Exp II: Overall Effect of Congestion on UE and BE. Figure 7 provides a comprehensive comparison between UE and BE with different levels of congestion using the same two designs as Figure 4 in §4.2. For the uncongested design (a), UE is even more efficient than BE throughout the routing process. For (b), BE performs worse in the beginning routing iterations but surpasses UE greatly in the following iterations when the congestion cases arise. We can observe that the efficacy of BE and UE is associated with the impact of bottleneck connections on runtime. In design (a) and the beginning iterations of (b), routing bottleneck connections is not the major runtime composition, and UE is more advantageous, while later iterations in (b) witness the dominance of bottleneck connections and the lead of BE.

Conclusions Drawn from the Observations. BE is a better candidate for intra-connection routing in congested conditions when bottleneck connections make up the majority of runtime, while UE is better at uncongested cases.

6 ADAPTIVE ROUTING ENHANCEMENT

To overcome the disadvantage of BE when routing non-bottleneck connections and maximize the performance of UE and BE, we design an adaptive strategy for intra-connection routing mode selection in §6.1. Then we elaborate on the seamless integration of our pluggable intra-connection routing approaches with inter-connection parallelism in §6.2.

6.1 Adaptive Intra-Connection Routing

We propose the adaptive intra-connection routing in view of the observations and conclusions in §5.3. The key is to put forward an *accurate*, *robust*, and *compute-efficient* metric that estimates whether BE or UE will perform better on a given connection. It is natural to predict the overall congestion condition the connection may undergo before routing it. Recent works on FPGA routing congestion prediction [1, 18, 29] forecast the regional congestion in the coming routing stage, mostly via vision-based machine learning (ML) models. However, these techniques provide coarse-grained, once-and-for-all congestion prediction on the FPGA device, while we require online, connection-level prediction in each routing iteration. Additionally, ML-based inferences may incur considerable computational cost and can be biased with new data in our scenario. In practice, we discover that estimating the difficulty to route the next connection by extracting relevant congestion features (*e.g.*, HPWL, density of overused nodes) exhibits large uncertainty and computational overhead.

Instead of adhering to congestion features, we propose a simple yet effective adaptive metric that considers the connection's routing history in previous iterations. From the previous analytical experiments, we observe that 1) the impact of bottleneck connections in beginning iterations tends to be small, and 2) if a connection is ripped up and rerouted in multiple iterations, the resource expansions and runtime tend to grow chronologically. The observation 1) is illustrated in Figure 4, and 2) is reasonable since congestion rises with routing proceeded. Such empirical conclusions aid us to design the following adaptive routing procedure:

- In the 1st iteration, route all connections with UE.
- In later iterations, for each connection *conn* to route:
    - If *conn* became bottleneck connection (*i.e.,* routing time $> 1ms$) in the last iteration it was routed, route *conn* with BE in this and later iterations.
    - Otherwise, route *conn* with UE.

In this way, we replace the dubious congestion prediction with a posterior, experience-based metric. Now our adaptive routing can effortlessly judge the exploration mode at connection level.

Table 1: Runtime and speedup of AceRoute with different intra-connection routing configurations, in respect to the baseline routers RWRoute and Vivado. The speedup and the average normalized runtime are relative to RWRoute. The designs are sorted by their routing time of RWRoute in ascending order.

	RWRoute	**Vivado**		**Ours-Basic (Serial)**		**Ours-BE (Serial)**		**Ours-BE (2-thread)**			**Ours-Adapt (Serial)**	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Runtime Var.	Speedup	Time (s)	Speedup
corescore_500	134.8	217.0	0.62×	107.1	1.26×	105.2	1.28×	69.7	(-4.2%, +1.6%)	1.93×	101.9	1.32×
rosetta_fd	150.7	153.3	0.98×	109.3	1.38×	80.6	1.87×	50.4	(-3.2%, +5.2%)	2.99×	81.1	1.86×
vtr_lu64peeng	192.8	216.5	0.89×	168.1	1.15×	147.9	1.30×	91.9	(-4.1%, +3.4%)	2.10×	137.8	1.40×
corundum_25g	230.2	——	——	187.0	1.24×	162.1	1.42×	106.8	(-6.9%, +12.3%)	2.16×	154.5	1.49×
vtr_mcml	235.3	469.2	0.50×	198.0	1.18×	77.0	3.06×	49.7	(-5.6%, +9.9%)	4.73×	72.6	3.24×
boom_med_pb	241.3	200.4	1.20×	185.7	1.31×	121.7	1.98×	82.6	(-8.2%, +9.1%)	2.92×	98.4	2.45×
corescore_500_pb	267.3	337.0	0.79×	203.2	1.32×	154.6	1.73×	106.5	(-4.8%, +14.6%)	2.51×	153.7	1.74×
koios_dla_large	322.7	665.0	0.49×	240.7	1.34×	311.1	1.04×	173.0	(-5.6%, +5.5%)	1.87×	237.4	1.36×
ispd16_example2	545.6	596.2	0.92×	411.6	1.32×	440.4	1.24×	338.2	(-4.7%, +6.2%)	1.61×	381.8	1.43×
boom_soc	1416.1	1026.1	1.38×	984.8	1.44×	749.8	1.89×	620.5	(-15.6%, +20.3%)	2.28×	647.5	2.19×
corescore_1700	1572.3	1100.2	1.43×	1097.3	1.42×	604.3	2.60×	447.6	(-2.7%, +4.4%)	3.51×	612.1	2.57×
mlcad_d181_left...	2008.5	878.4	2.29×	622.7	3.22×	393.6	5.10×	283.4	(-5.8%, +8.2%)	7.09×	375.2	5.35×
mlcad_d181	4555.1	4028.7	1.13×	5111.1	0.90×	980.3	4.65×	882.9	(-6.2%, +6.4%)	5.16×	954.5	4.77×
boom_soc_v2	5869.7	2024.6	2.90×	3866.1	1.52×	1504.7	3.90×	1135.2	(-6.3%, +10.1%)	5.17×	1344.2	4.37×
Average	1.000	1.080	1.19×	0.754	1.43×	0.539	2.36×	0.368	(-6.0%, +8.4%)	3.29×	0.491	2.55×

*We run the BE (2-thread) router for 8 times, recording the mean time and the variation range relative to the mean of 8 runs. Other deterministic routers are averaged with 3 runs.

**Vivado aborts when routing `corundum_25g` due to DRC errors.

6.2 Inter-Connection Parallelism Integration

Due to the orthogonality to connection ordering, our adaptive intra-connection routing scheme can be smoothly integrated into existing inter-connection parallel routers. Furthermore, the adaptive routing augments typical partition-based parallelism. Current parallel routers suffer from the problem of load balance. The workload assigned to each process or thread is inevitably unbalanced, considering the uneven distribution of congestion. In a device region of high connection density, partitioning the connections into disjoint sets is almost impossible. Traditional UE-based routing is prone to congestion, causing inequality among the partitions with different levels of bottleneck connections. On the contrary, our adaptive routing handles bottleneck and non-bottleneck connections using BE and UE respectively with close runtime, which flattens the workload among different connection groups.

We implement recursive bi-partitioning [4, 16], a common inter-connection parallelism scheme. The device region is divided into two sub-regions with a median cutline. This creates three subgroups of connections: 1) connections entirely in the first sub-region, 2) connections entirely in the second sub-region, and 3) connections across the two sub-regions. The first two sub-groups are independent and can be routed in parallel, while the third group is then routed sequentially. These steps can be repeated recursively for more parallelism. However, the poor scalability restricts the speedup since groups 1) and 2) contain fewer connections in deeper partitions, while 3) may include many bottleneck connections. Adaptive routing can alleviate this imbalance and provide extra speedup for recursive bi-partitioning.

7 EXPERIMENTS

The general experimental setup has been presented in §3, while some details will be further clarified in this section. §7.1 evaluates the performance of AceRoute by comparisons with RWRoute [30] and Vivado [2]. In §7.2, we measure the acceleration when integrating inter-connection parallelism with intra-connection approaches.

7.1 Intra-Connection Routing Optimizations

As for intra-connection routing, we evaluate AceRoute of the following configurations: **1) Basic**, PathFinder-based version with UE and serial implementation, **2) BE (Serial)**, implementing serial BE for all connections (§5.1), **3) BE (2-thread)**, the parallel version of 2) with simply 2 threads (§5.2), and **4) Adapt (Serial)** with our adaptive strategy to switch between serial BE and UE (§6.1). Note that we do not evaluate the adaptive version with parallel BE because it will always run slower than 3) considering that the worst case for 2-thread BE is UE for each connection, and thus, we focus on the serial adaptive implementation with determinism.

Table 1 displays the runtime and speedup results of RWRoute, Vivado, and our router of different configurations. The speedup is relative to RWRoute, and the bottom row specifies the average of normalized runtime and speedup with RWRoute. The rows are sorted by the RWRoute runtime. Since the BE (2-thread) version does not guarantee determinism, we run it for *8 times* independently, recording its runtime with the mean value and the range of variation relative to the mean. All other configurations are serial and promise determinism. Additionally, Table 2 summarizes the normalized critical-path wirelength with RWRoute, with the min, max, and mean values among all the designs in the benchmark. For BE (2-thread), the results are averaged among multiple runs.

Results in Table 1 and Table 2 demonstrate that our renewed intra-connection routing techniques achieve significant acceleration with minor wirelength overheads. Our baseline routers, RWRoute and Vivado, show similar performance over the benchmark. Our basic router has a slight runtime gain compared with RWRoute, which is not essential and mainly attributed to the language shift from Java to C++ given the similar algorithm framework. For pure BE intra-connection routing, the serial and 2-thread parallel implementations exhibit an average 2.36× and 3.29× speedup, respectively. The efficient 2-thread implementation of parallel BE introduces little parallelism overhead and acceptable fluctuations of runtime among multiple runs. For some designs insensitive to bottleneck connections, *e.g.,* `ispd16_example2` and `koios_dla_large`, the runtime of serial BE deteriorates in regard to the basic UE version. Our serial adaptive intra-connection router achieves 2.55× and 2.48× speedup on average, compared with RWRoute and Vivado, respectively. The adaptive router improves the serial BE router with up to 25% runtime reduction on `koios_dla_large`, and an average 10% runtime reduction over all designs. Furthermore, the results highlight the ***scalability*** of AceRoute given that the speedup of BE (2-thread) and adaptive routing generally becomes more prominent on complex designs that take more time to route with RWRoute. The underlying reason could be the heavier congestion and larger impact of bottleneck connections in these designs.

Table 2: Critical-path wirelength normalized with RWRoute across all designs (min/max/mean) for different routers.

		Min	Max	Mean
Base Routers	RWRoute	1.00	1.00	1.00
	Vivado	0.83	1.51	1.12
Intra-Connection Opt. Only	Basic	0.91	1.23	1.03
	BE (Serial)	0.82	1.26	1.02
	BE (2-thread)	0.89	1.62	1.08
	Adaptive	0.92	1.25	1.04
Inter-Connection Par. × Intra-Connection Opt.	Basic	0.83	1.64	1.03
	BE (2-thread)	0.90	1.26	1.04
	Adaptive	0.82	1.27	1.01

7.2 Enhanced Inter-Connection Parallelism

To verify the extra performance gain over inter-connection parallelism brought by our intra-connection optimizations, we integrate our **1) basic, serial**, **2) BE (2-thread)**, and **3) serial adaptive** approaches into an inter-connection parallel routing framework based on recursive bi-partition. Table 3 shows the speedup results relative to RWRoute. The recursive bi-partition parallel routing is deterministic since the order of connections to route is predetermined. Like Table 1, only BE (2-thread) enhanced routing compromises determinism and is evaluated with 8 independent runs. We use 9 threads for bi-partition-based routing as its mechanism requires 3^n parallelism. The theoretically maximum thread utilization is thus 18 with BE (2-thread), and 9 for basic and adaptive routing.

BE (2-thread) and adaptive routing drive the bi-partition speedup from 2.22× to 5.01× and 4.36× on average, respectively. Bi-partition parallelism suffers from complex designs with numerous conflicting connections. On `mlcad_d181_lefttwo3rds`, the original partition-based parallelism is even slower than the serial version, with a speedup of 1.59× in contrast to 3.22×. Once BE (2-thread) or adaptive techniques are plugged in, the speedup rises to 9.80× and 10.23×, respectively. The overall performance of intra-connection serial adaptive routing is close to the 2-thread BE, demonstrating the capacity of our adaptive strategy to boost inter-connection parallel routers.

Table 3: Accelerated inter-connection parallelism by intra-connection optimizations, with speedup over RWRoute.

Bi-partition Par. ×	Basic	BE (2-thread)		Adapt
	Speedup	Runtime Var.	Speedup	Speedup
corescore_500	2.86×	(-4.3%, +3.8%)	3.65×	2.93×
rosetta_fd	1.46×	(-3.6%, +4.1%)	3.61×	2.77×
vtr_lu64peeng	2.45×	(-8.8%, +7.7%)	3.53×	2.74×
corundum_25g	1.90×	(-1.3%, +3.8%)	2.75×	2.21×
vtr_mcml	2.25×	(-4.0%, +5.8%)	7.17×	5.23×
boom_med_pb	1.41×	(-5.8%, +9.1%)	3.48×	2.87×
corescore_500_pb	2.57×	(-12.0%, +8.3%)	4.11×	3.47×
koios_dla_large	3.57×	(-1.1%, +1.4%)	4.25×	3.31×
ispd16_example2	3.01×	(-6.4%, +12.7%)	2.61×	2.63×
boom_soc	1.61×	(-11.7%, +10.8%)	3.93×	3.74×
corescore_1700	3.26×	(-3.5%, +3.9%)	8.02×	6.64×
mlcad_d181_left...	1.59×	(-6.1%, +5.5%)	9.80×	10.23×
mlcad_d181	1.23×	(-8.0%, +11.8%)	6.13×	6.09×
boom_soc_v2	1.94×	(-5.8%, +4.7%)	6.90×	6.19×
Average	2.22×	(-5.9%, +6.7%)	5.01×	4.36×

*Like Table 1, BE (2-thread) integrated router has 8 runs, and others 3 runs.

8 CONCLUSION

In this work, we first point out the two levels of granularity in mainstream FPGA routing: intra-connection and inter-connection. We profile this routing paradigm in depth and discover that resolving bottleneck connections in congested designs dominates the runtime. To mitigate the resource expansions and reduce routing time, we revolutionize the classical A*-based intra-connection routing with bidirectional path exploration for the first time. Nevertheless, through dedicated case studies, we observe that unidirectional intra-connection routing prevails in uncongested conditions. Accordingly, we propose a simple yet effective adaptive strategy for real-time exploration mode determination of a connection based on its routing time in previous iterations. Moreover, the pluggability of our approach allows flexible and effortless integration of adaptive bidirectional search into existing FPGA routing frameworks.

The experimental results demonstrate a significant speedup of AceRoute over established academic and industrial routers on various designs with different sizes and complexity. Furthermore, intra-connection adaptive bidirectional exploration overcomes the deficiencies of inter-connection parallel routers and magnifies their performance after integration. This work represents a remarkable step forward in FPGA routing acceleration, offering both theoretical insights and practical solutions for tackling the challenge of FPGA routing in large, congested designs. Our future work includes developing efficient deterministic parallel bidirectional strategies, improving the dual cost design in bidirectional search with better awareness of FPGA architecture, and optimizing the adaptivity for robuster routing mode selection.

ACKNOWLEDGMENTS

We are grateful to Eddie Hung and all other organizers of Runtime-First FPGA Interchange Routing Contest @ FPGA'24 for the development of routing benchmarks and baseline tools.

This work was partly supported by the National Key R&D Program of China (Grant No. 2022YFB4500500) and the National Natural Science Foundation of China (Grant No. 62090021).

REFERENCES

[1] Mohamed Baker Alawieh, Wuxi Li, Yibo Lin, Love Singhal, Mahesh A Iyer, and David Z Pan. 2020. High-definition routing congestion prediction for large-scale FPGAs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 26–31.

[2] AMD/Xilinx. 2023. Overview of AMD Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[3] Shih-Chun Chen and Yao-Wen Chang. 2017. FPGA placement and routing. In *International Conference on Computer-Aided Design (ICCAD)*. 914–921.

[4] Marcel Gort and Jason H Anderson. 2011. Accelerating FPGA routing through parallelization and engineering enhancements special section on PAR-CAD 2010. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 31, 1 (2011), 61–74.

[5] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[6] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. 67–76.

[7] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 6 (2021), 1303–1320.

[8] Eddie Hung, Chris Lavin, Zak Nafziger, and Alireza Kaviani. 2024. Runtime-First FPGA Interchange Routing Contest @ FPGA'24. https://xilinx.github.io/fpga24_routing_contest/index.html.

[9] Mike Hutton and Vaughn Betz. 2018. FPGA synthesis and physical design. In *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC Press, 13–1.

[10] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling custom crafted implementations for FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–140.

[11] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A negotiation-based performance-driven router for FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. 111–117.

[12] Yehdhih Ould Mohammed Moctar and Philip Brisk. 2014. Parallel FPGA routing based on the operator formulation. In *Design Automation Conference (DAC)*. 1–6.

[13] Wim Pijls and Henk Post. 2009. A new bidirectional search algorithm with shortened postprocessing. *European Journal of Operational Research* 198, 2 (2009), 363–369.

[14] Ira Pohl. 1969. *Bi-directional and heuristic search in path problems*. Technical Report. Stanford Linear Accelerator Center (SLAC), Menlo Park, CA (United States).

[15] Luis Henrique Oliveira Rios and Luiz Chaimowicz. 2011. PNBA*: A parallel bidirectional heuristic search algorithm. In *ENIA VIII Encontro Nacional de Inteligê ncia Artificial*.

[16] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *International Conference on Computer-Aided Design (ICCAD)*. 118–125.

[17] Shashwat Shrivastava, Stefan Nikolić, Chirag Ravishankar, Dinesh Gaitonde, and Mirjana Stojilović. 2023. IIBLAST: Speeding Up Commercial FPGA Routing by Decoupling and Mitigating the Intra-CLB Bottleneck. In *International Conference on Computer-Aided Design (ICCAD)*. 1–9.

[18] Umair Siddiqi, Timothy Martin, Sam Van Den Eijnden, Ahmed Shamli, Gary Grewal, Sadiq Sait, and Shawki Areibi. 2022. Faster fpga routing by forecasting and pre-loading congestion information. In *Workshop on Machine Learning for CAD (MLCAD)*. 15–20.

[19] Lenie Sint and Dennis de Champeaux. 1977. An improved bidirectional heuristic search algorithm. *Journal of the ACM (JACM)* 24, 2 (1977), 177–191.

[20] Mirjana Stojilović. 2017. Parallel FPGA routing: Survey and challenges. In *International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[21] Frank Vahid, Greg Stitt, and Roman Lysecky. 2008. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer* 41, 7 (2008), 40–46.

[22] Elias Vansteenkiste, Karel Bruneel, and Dirk Stroobandt. 2013. A connection-based router for FPGAs. In *International Conference on Field-Programmable Technology (FPT)*. 326–329.

[23] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2019. CRoute: A fast high-quality timing-driven connection-based FPGA router. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 53–60.

[24] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. 2019. ParRA: A shared memory parallel FPGA router using hybrid partitioning approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 4 (2019), 830–842.

[25] Dekui Wang, Jun Feng, Yaqiong Xing, Ke Liu, Wei Zhou, Xingxing Hao, Xiaodan Zhang, and Lili Zhang. 2023. A Fast FPGA Connection Router Using Pre-routing Based Parallel Local Routing Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 42, 11 (2023), 3868–3880.

[26] Pengkai Wang, Husna Mutahira, Jonghoek Kim, and Mannan Saeed Muhammad. 2023. ABA*–Adaptive Bidirectional A* Algorithm for Aerial Robot Path Planning. *IEEE Access* (2023).

[27] Dawen Xia, Bingqi Shen, Yongling Zheng, Wenyong Zhang, Dewei Bai, Yang Hu, and Huaqing Li. 2024. A bidirectional-a-star-based ant colony optimization algorithm for big-data-driven taxi route recommendation. *Multimedia Tools and Applications* 83, 6 (2024), 16313–16335.

[28] Xinshi Zang, Wenhao Lin, Shiju Lin, Jinwei Liu, and Evangeline FY Young. 2024. An open-source fast parallel routing approach for commercial FPGAs. In *Great Lakes Symposium on VLSI (GLSVLSI)*. 164–169.

[29] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *Design, Automation, and Test in Europe (DATE)*. 1130–1135.

[30] Yun Zhou, Pongstorn Maidee, Chris Lavin, Alireza Kaviani, and Dirk Stroobandt. 2021. RWRoute: An open-source timing-driven router for commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 1 (2021), 1–27.

[31] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. 2020. Accelerating FPGA routing through algorithmic enhancements and connection-aware parallelization. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 4 (2020), 1–26.