An Open-Source Fast Parallel Routing Approach for Commercial FPGAs

Xinshi Zang
The Chinese University of Hong Kong
xszang@cse.cuhk.edu.hk

Wenhao Lin
The Chinese University of Hong Kong
whlin23@cse.cuhk.edu.hk

Shiju Lin
The Chinese University of Hong Kong
sjlin@cse.cuhk.edu.hk

Jinwei Liu
The Chinese University of Hong Kong
jwliu@cse.cuhk.edu.hk

Evangeline F.Y. Young
The Chinese University of Hong Kong
fyyoung@cse.cuhk.edu.hk

ABSTRACT

In the face of escalating complexity and size of contemporary FPGAs and circuits, routing emerges as a pivotal and time-intensive phase in FPGA compilation flows. In response to this challenge, we present an open-source parallel routing methodology designed to expedite routing procedures for commercial FPGAs. Our approach introduces a novel recursive partitioning ternary tree to augment the parallelism of multi-net routing. Additionally, we propose a hybrid updating strategy for congestion coefficients within the routing cost function to accelerate congestion resolution in negotiation-based routing algorithms. Evaluation on public benchmarks from the FPGA24 routing contest demonstrates the efficacy of our parallel router. It achieves a 2× speedup compared to the academic serial router RWRoute. Furthermore, when compared to the industry-standard tool Vivado, our approach not only delivers a 2× acceleration but also yields a notable 31% enhancement in critical-path wirelength.

ACM Reference Format:
Xinshi Zang, Wenhao Lin, Shiju Lin, Jinwei Liu, and Evangeline F.Y. Young. 2024. An Open-Source Fast Parallel Routing Approach for Commercial FPGAs. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649476.3658714

1 INTRODUCTION

Routing constitutes a pivotal phase in the compilation flow of Field-Programmable Gate Arrays (FPGAs), entailing the assignment of each net to wire segments on FPGAs to ensure connectivity without overlap. With the escalating complexity of modern FPGAs and circuits, routing has evolved into an exceedingly challenging and time-consuming process [1]. Accelerating routing procedures holds profound significance across various FPGA application domains, such as ASIC emulation.

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14218422).

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA

ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3658714

Numerous existing works have endeavored to expedite FPGA routing runtime through parallel routing techniques employing multi-threads and Graphics Processing Units (GPUs) [2, 3, 7–11, 14]. Multi-net parallelization is central to these efforts [2, 3, 8, 10, 11, 14], wherein different recursive spatial partitioning methods are employed to divide nets into geographically independent sets, facilitating parallel routing. Additionally, some studies have explored single-net parallelization by parallelizing maze expansion or leveraging GPU-accelerated Bellman-Ford algorithms [7, 9]. While significant acceleration has been achieved, the majority of existing parallel FPGA routing works are not open source, hindering community development. Notably, VTR [8] is a popular open-source FPGA CAD tool; however, it primarily targets academic hypothetical architectures, rendering it impractical for commercial FPGAs.

Addressing these limitations, we propose an open-source parallel FPGA routing approach tailored for commercial devices. We leverage the open-source negotiation-based serial router, RWRoute [13], which is contained in the RapidWright [5] framework and is compatible with commercial FPGAs. Different from the binary partitioning tree used in existing works [2, 3, 14], we propose a novel recursive partitioning ternary tree (RPTT) to schedule the parallel routing of multiple nets. We iteratively construct a full ternary tree for each set of nets within an FPGA region until further partitioning is unfeasible. By enabling independent routing of nets in the left and right RPTT sub-trees, our RPTT significantly enhances the parallelization of multi-net routing. Furthermore, we propose a Hybrid Updating Strategy (HUS), incorporating present-centric and historical-centric updating, for congestion coefficient adjustments within the routing cost function. This hybrid strategy effectively balances pathfinding time and congestion resolution, particularly in congested designs. The specific contributions of this work are summarized as follows:

- We propose an open-source[1] parallel FPGA routing approach to expedite routing for commercial FPGAs.
- We introduce a Recursive Partitioning Ternary Tree (RPTT) to enhance multi-net parallelization and a Hybrid Updating Strategy (HUS) to accelerate congestion resolution.
- On the FPGA24 contest public benchmarks [4], our parallel router outperforms the serial router RWRoute by achieving approximately twice the speed. Furthermore, our router surpasses the commercial tool Vivado, exhibiting a 31% improvement in critical-path wirelength and doubling the speed.

[1]The codes are available at https://github.com/xszang/parallel-routing.

The remainder of this paper is organized as follows: Section 2 presents preliminaries about the commercial UltraScale+ FPGA architecture. Section 3 discusses our partitioning-based parallel routing framework with the hybrid updating strategy. Section 4 provides detailed experimental results and ablation studies. Finally, Section 5 concludes the work.

2 PRELIMINARIES

In this section, we provide an overview of the target FPGA architecture, the routing resource graph, and discuss fundamental serial FPGA routing algorithms.

2.1 Ultrascale+ FPGA Architecture

We focus on Xilinx Virtex UltraScale+ FPGAs in this study, characterized by a columnar architecture as depicted in Fig. 1. This architecture comprises configurable logic blocks (CLB), block RAM (BRAM), DSP units, IO interfaces, and interconnect (INT) blocks. The INT blocks are interspersed among other functional blocks, forming a programmable interconnection network that facilitates connections between various functional blocks. The wires interconnecting INT blocks are fixed routing resources and can span 1, 2, 4, or 12 INT blocks. Notably, the INT block is configurable, allowing for diverse wire connectivities. Fig. 2 illustrates a simplified INT block example, with dashed lines indicating programmable interconnect points (PIPs).

2.2 Routing Resource Graph

Routing resources encompass wires and PIPs on the FPGA device, typically represented as a directed graph $G = (V, E)$. In the routing resource graph (RRG), nodes correspond to wires, and edges represent PIPs. Fig. 2 illustrates the construction of a routing resource graph. Each RRG node's length is proportional to the number of INT blocks spanned by the corresponding wire. Compared to academic hypothetical architectures, the UltraScale+ FPGA exhibits an extensive routing resource graph, comprising over 28 million nodes and 125 million edges, imposing significant complexity on routing tasks.

After placement, the signal nets that have no direct wires connecting them need to be routed on the RRG. The source and sink pins of nets are first mapped to RRG nodes. The routing problem is then transformed into a path-searching task on the RRG. For each net, the source and sink pins need to be connected by using as few RRG nodes as possible. Each RRG node can only accommodate at most one net, or in other words, the capacity of a RRG node for routing is 1. The resource overflows, meaning that multiple nets pass through one RRG node, are prohibited in FPGA routing.

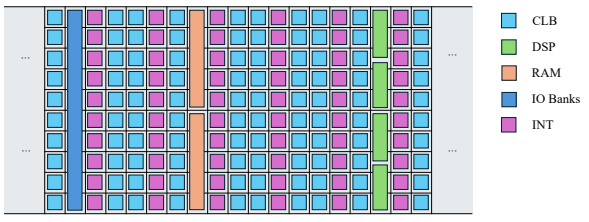
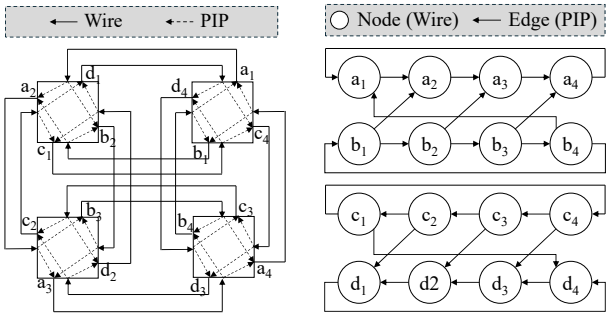


Figure 1: UltraScale+ FPGA with columnar resources



(a) Interconnect architecture **(b) Routing graph**

Figure 2: Construction of a routing resource graph

2.3 FPGA routing algorithm

The negotiation-based routing algorithm, exemplified by PathFinder [6], is pivotal and widely employed in FPGA routers. PathFinder employs a negotiation mechanism, defining the cost function of a given node n as shown in Eq. (1),

$$f(n) = (b(n) + h(n)) \times p(n) \tag{1}$$

where $b(n)$ denotes the base cost associated with the node's length, $h(n)$ represents accumulated historical congestion cost, and $p(n)$ signifies present congestion cost. By incrementally increasing congestion costs, the routing algorithm eventually finds legal routing solutions for all nets without congestion.

Existing routing algorithms can be categorized into net-based [8, 10, 11] and connection-based [3, 13, 14] routers, depending on their treatment of multi-pin nets. Net-based algorithms route entire multi-pin nets as a whole, while the connection-based routers first split a multi-pin net into several two-pin nets (called connections) and then route each connection independently. To minimize net wirelength, connection-based routers encourage reuse of common RRG nodes among different connections from the same net. As the overlap between connections is typically smaller than that between nets, connection-based routing algorithms enable greater parallelism for parallel routing. Consequently, we adopt the connection-based approach in this work to develop the parallel algorithm.

3 METHODOLOGY

In this section, we will describe the framework of our parallel router in detail, as depicted in Fig. 3. Given the placement solution, we first build a recursive partitioning ternary tree (RPTT) to facilitate the scheduling of multi-net parallel routing. Subsequently, we iteratively perform parallel routing based on the RPTT and update the congestion coefficients until a valid and resource overflow-free routing solution is found. Additionally, we employ a hybrid updating strategy (HUS), containing present-centric and historical-centric updating, for congestion coefficients at the end of each routing iteration to expedite congestion resolution.

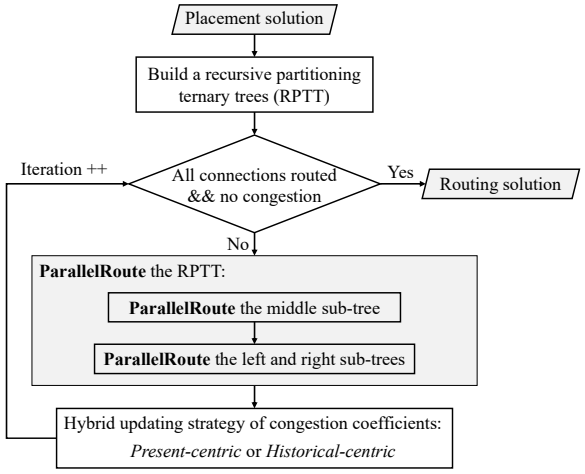


Figure 3: The framework of our parallel router

3.1 Recursive Partitioning Ternary Tree (RPTT)

Partitioning-based parallel routing is widely adopted by existing parallel routing works [2, 3, 10]. The bi-partitioning approach is first applied to recursively split the FPGA board into two disjoint regions with interleaved horizontal and vertical cutlines. A full binary partitioning tree is then built where each sub-tree represents the nets fully contained in a region and the root node of a sub-tree represents the nets crossing the cutline. With the partitioning tree, the routing of all nets is scheduled by first routing the root node and then parallelly routing the two sub-trees.

To further speed up the routing of the root node and establish a universal data structure for scheduling, we propose a recursive partitioning ternary tree (RPTT) in this work, which is described in the function (line 6) of Alg. 1. The left and right sub-trees of RPTT are similar to those of the aforementioned binary partitioning tree but RPTT also constructs a middle sub-tree for the nets crossing the cutline. The detailed procedure to determine the cutline in line 7 of Alg. 1 will be discussed later. The connections on each non-leaf node in RPTT represent the union of connections in its sub-trees. To demonstrate the RPTT build process, one RPTT is provided in Fig. 5 for the connections distributed in Fig. 4.

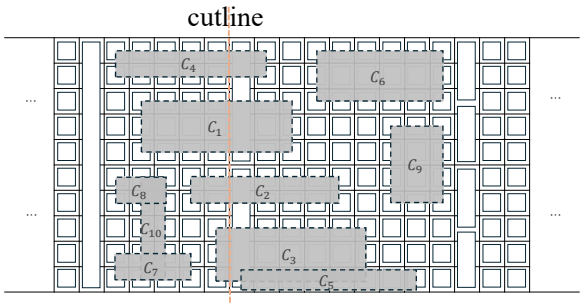


Figure 4: Ten connections with one cutline

Algorithm 1: RPTT-based Parallel Routing

Input: Connections C

1 Initialize an empty ternary tree T
2 `BuildTree`(C, $T.root$)
3 Initialize an empty ThreadPool *pool*
4 `ParallelRoute`(*root*, *pool*)
5 Finish all works in *pool*
6 **Function** `BuildTree`(*Connections* C, *Tree root*):
7 $success, C_l, C_m, C_r \leftarrow$ `BalanceCut`(C)
8 **if** *success* **then**
9 $root.left \leftarrow$ `BuildTree`(C_l, $root.left$)
10 $root.mid \leftarrow$ `BuildTree`(C_m, $root.mid$)
11 $root.right \leftarrow$ `BuildTree`(C_r, $root.right$)
12 **else**
13 Set $root.left$, $root.mid$, and $root.right$ as *null*
14 $root.connections \leftarrow C$
15 **return** *root*
16 **Function** `ParallelRoute` (*RPTT root, ThreadPool pool*):
17 **if** *root has sub-trees* **then**
18 `BlockedRoute`($root.mid$)
19 *pool.add*(`ParallelRoute`($root.left$, *pool*))
20 *pool.add*(`ParallelRoute`($root.right$, *pool*))
21 **else**
22 Sequentially route connections in $root.connections$
23 **Function** `BlockedRoute` (*RPTT root*):
24 Initialize an empty ThreadPool *pool*
25 `ParallelRoute`(*root*, *pool*)
26 Finish all works in *pool*

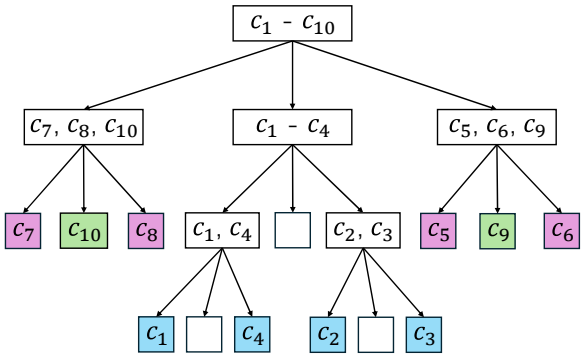


Figure 5: RPTT for the connections in Fig. 4

After building the RPTT, we then perform a universal parallel routing procedure in line 4 of Alg. 1. The `ParallelRoute` is a recursive function that first performs the routing of the middle sub-tree in a blocked way and then submits two parallel jobs of the routing of the left and right sub-trees. The `BlockedRoute` requires that all routing tasks submitted in this sub-trees need to be finished at the end of the function while the `ParallelRoute` doesn't have this requirement. A unique task pool is maintained for each middle

Algorithm 2: Balance-driven Cutline

Function `BalanceCut` *(Connections C)*:
  $success^x, C_l^x, C_m^x, C_r^x, diff^x \leftarrow$ `BalanceCutX`(C)
  $success^y, C_l^y, C_m^y, C_r^y, diff^y \leftarrow$ `BalanceCutY`(C)
  if $diff^x < diff^y$ **then**
    return $success^x, C_l^x, C_m^x, C_r^x$
  else
    return $success^y, C_l^y, C_m^y, C_r^y$
Function `BalanceCutX`(*Connections C*):
  get the horizontal boundary (x_{min}, x_{max}) of C
  Initialize lists $size\_before$ and $size\_after$ with 0
  for *Connection* $c \in C$ **do**
    for $i \in [c.x_{max}, x_{max}]$ **do**
      $size\_before[i] \leftarrow size\_before[i] + 1$
    for $i \in [x_{min}, c.x_{min})$ **do**
      $size\_after[i] \leftarrow size\_after[i] + 1$
  $diff \leftarrow inf, cutline \leftarrow 0$
  for $i \in [x_{min}, x_{max}]$ **do**
    $d \leftarrow |size\_after[i] - size\_before[i]|$
    if $d < diff$ **then**
      $diff \leftarrow d, cutline \leftarrow i$
  $C_l \leftarrow \{c \in C | c.x_{max} \leq cutline\}$
  $C_r \leftarrow \{c \in C | c.x_{min} > cutline\}$
  $C_m \leftarrow \{c \in C | c \notin C_l, C \notin C_r\}$
  $success \leftarrow true$
  if C_l *is empty or* C_r *is empty* **then**
    $success \leftarrow false$
  return $sucess, C_l, C_m, C_r, diff$
Function `BalanceCutY`(*Connections C*):
  `/* Similar with BalanceCutX, skipped here */`

sub-tree to manage the routing jobs which are independent and invoked recursively on the sub-tree. With the scheduling method in Alg. 1, the execution orders of different connections in Fig. 5 are marked with different colors. The four blue connections will be routed firstly in parallel and then the two green ones follow and the four pink ones are parallelly routed at last.

To build an efficient partitioning tree for parallelism, it is of great importance to select a good cutline that splits the nets in balance. To this end, we propose a balance-driven algorithm inspired by ParaDRo [3] to determine the optimal cutline for balanced partitioning. As described in Alg. 2, we will try to do the partitioning on the x-axis and y-axis and adopt the one with a smaller difference in partition size. Whatever the axis is, two lists, including $size\_bebore$ and $size\_after$, are maintained to accumulate the connection numbers aside from a cutline. By doing so, the optimal cutline with the smallest imbalance in partition size can be found in a linear time. The recursive partitioning will be terminated in a region if no cutline can split the connections into two independent parts.

3.2 Hybrid Updating Strategy (HUS)

In traditional path-finding algorithms, the A^* searching algorithm is usually first applied to find the short path between the source node and the sink node for each net. The path length during A^* searching is related to both wirelength and the congestion cost. By gradually increasing the congestion penalty, the routing algorithms can converge to a congestion-free solution with optimized wirelength. However, as pointed out in the enhanced PathFinder algorithm [12], the updating strategy of the congestion penalty has a great influence on both quality and runtime. In this work, a hybrid updating strategy (HUS) is proposed to speed up the congestion fixing during the negotiation-based routing algorithm.

Before discussing the HUS, we first introduce the routing cost function of RRG nodes in this work. As defined in Eq (2), the node cost, $f(n)$, is composed of three parts. The first term is the upstream path length, represented by $c_{prev}(n)$, which is the accumulated cost of all nodes in the partial path from the source to the node n. The second $c_{est}(n)$ denotes the estimation of the cost of reaching the sink from the node n, which is estimated by the Mahattan distance.

$$f(n) = c_{prev}(n) + c_{est}(n) + c(n) \tag{2}$$

The third item, $c(n)$ in Eq. (2), defines the cost of using the node n, which is further formulated in Eq. (3). $c(n)$ depends on the wirelength-related base cost $b(n)$, the present congestion cost $p(n)$ and the historically accumulated congestion cost $h(n)$. $share(n)$ represents the number of connections from the same net that are using n, encouraging connections to reuse the existing path of the net.

$$c(n) = \frac{b(n) \cdot h(n) \cdot p(n)}{1 + share(n)} \tag{3}$$

To resolve routing congestions, $h(n)$ and $p(n)$ are increased with different updating strategies during the entire routing process. The updating strategies in RWRoute are defined in Eq. (4) and (5), where $occ(n)$ is the number of nets using RRG node n and i represents the number of iterations of routing all nets. h_f and p_f are const congestion coefficients (1 and 2 respectively) in RWRoute. The historical cost $h(n)$ is linear with h_f while the present cost $p(n)$ is exponential to p_f.

$$h^i(n) = \begin{cases} h^{i-1}(n) + h_f \cdot (occ(n) - 1) & \text{, if } occ(n) > 1 \\ h^{i-1}(n) & \text{, otherwise} \end{cases} \tag{4}$$

$$p^i(n) = 1 + p_0 \cdot (p_f)^{i-1} \cdot occ(n) \tag{5}$$

Table 1: Statistics of FPGA24 public benchmarks

Benchmark	Nets (k)	Connections (k)	LUTs (k)	FFs (k)	DSPs	BRAMs
logicnets_jscl	28	180	31	2	0	0
boom_med_pb	54	221	36	17	24	142
vtr_mcml	71	225	43	15	105	142
rosetta_fd	77	230	46	39	72	62
corundum_25g	166	495	73	96	0	221
finn_radioml	110	405	74	46	0	25
vtr_lu64peeng	143	537	90	36	128	303
corescore_500	179	590	96	116	0	250
corescore_500_pb	175	597	96	116	0	250
mlcad_d181_lefttwo3rds	361	916	155	203	1344	405
koios_dla_like_large	509	912	189	362	2209	192
boom_soc	274	1374	227	98	61	161
ispd16_example2	449	1455	289	234	200	384
UltraScale+ xcvu3p	-	-	394	788	2280	720

Table 2: Overall performance. All metrics are the smaller the better.

Benchmark	Vivado			RWRoute			Ours		
	Runtime (s)	Wirelength	Score	Runtime (s)	Wirelength	Score	Runtime (s)	Wirelength	Score
logicnets_jscl	78.33	310	101.50	52.03	**226**	69.43	**35.26**	234	**55.13**
boom_med_pb	**139.33**	823	**207.70**	230.88	969	304.69	144.50	**806**	210.65
vtr_mcml	490.33	666	507.90	243.13	594	278.22	**94.29**	**584**	**143.26**
rosetta_fd	147.67	888	221.70	161.30	839	229.07	**125.32**	**804**	**193.19**
corundum_25g	-	-	-	249.61	**396**	264.25	**131.11**	500	**168.00**
finn_radioml	154.67	338	173.00	119.88	277	135.59	**63.29**	**251**	**82.06**
vtr_lu64peeng	218.67	1728	369.60	226.57	1412	345.12	**114.12**	**1333**	**236.01**
corescore_500	188.33	751	244.60	158.84	680	210.96	**73.03**	**668**	**132.52**
corescore_500_pb	226.67	861	290.10	278.30	**687**	319.17	**138.63**	739	**198.67**
mlcad_d181_lefttwo3rds	**407.67**	1159	482.80	1,779.59	809	1,682.53	409.81	**771**	**445.93**
koios_dla_like_large	542.33	927	580.80	392.07	548	407.67	**181.47**	**520**	**215.33**
boom_soc	711.00	2235	863.40	1,292.74	1698	1,333.26	**635.33**	**1673**	**739.10**
ispd16_example2	385.00	1481	494.60	584.94	1114	637.85	**314.65**	**939**	**377.09**
Avg. Ratio	2.04	1.31	1.73	2.10	1.03	1.76	1.00	1.00	1.00

*Vivado fails to route the corundum_25g due to the failure in the DRC during the routing.

The enhanced PathFinder [12] found that the quick growing speed of $p(n)$ will result in the net-order dependence issue and the original PathFinder algorithm can be improved by updating the historical congestion cost $h(n)$ alone. Hence, in the enhanced PathFinder, p_f and h_f are set to be 1 and 5 respectively. However, this setting will cause much more routing iterations to fix all routing conflicts. The maximum number of iterations in the enhanced PathFinder is set to 1500 to ensure successful routing for congested designs, which is much bigger than 50, the default setting of VTR [8].

Different from [12], we propose a hybrid updating strategy (HUS) for the historical and present coefficients to reduce the runtime of fixing congestion while improving the routing wirelength for congested designs. In this work, we regard a circuit to route as a congested design if the ratio of the number of congested RRG nodes to the number of connections after the first iteration is larger than a threshold. In HUS, we first perform the present-centric updating by keeping the original fast updating strategy for the present coefficient to finish the routing of those uncongested areas. After a few iterations, most of the remaining connections to route are located in congested areas. We then apply the second historical-centric updating for the historical coefficient by decreasing the p_f from 2 to α and increasing the h_f from 1 to β. In the work, α and β are set to 1.1 and 2 respectively. With the historical-centric updating, the negative impact of connection routing orders can be reduced, decreasing the number of iterations to fix the routing congestions.

4 EXPERIMENTAL RESULTS

In this section, we will analyze the experimental results of different methods including Vivado and RWRoute [13]. We followed the setting of FPGA24 routing contest [4] and conducted experiments in a high-performance computing server equipped with a 2.90GHz CPU and 768GB memory. Our approach was implemented in Java and ran with 16 threads by default. Vivado v2021.1 was compared with the command "`route_design -no_timing_driven -preserve`"[2] to conduct the routing task. The evaluation metrics of the FPGA24 contest include the runtime and critical wirelength. The overall score is equal to 0.9 * runtime + 0.1 * critical-path wirelength. We ran all experiments three times and showed the average of the results to reduce the randomness effect.

4.1 Benchmarks

The statistics of FPGA24 public benchmarks are summarized in Tab. 1. In FPGA24 contest, these circuits are obtained from different public benchmark suites and are then synthesized, placed, and routed on the target FPGA by using Vivado. The routing solutions of all signal nets are removed for the contest task. The benchmarks use the open-source FPGA Interchange Format (FPGAIF). The nets in Tab. 1 include all signal nets to be routed and the connections represent the corresponding two-pin sub-nets to be routed.

4.2 Overall Performance

The overall results of different methods are presented in Tab. 2. Compared with Vivado, RWRoute can significantly reduce the wirelength but incur considerable time overhead in some circuits, like mlcad_d181_lefttwo3rds and boom_soc. Compared with both Vivado and RWRoute, our router can not only run two times faster on average but also further improve the wirelength in most cases, demonstrating the effectiveness of our proposed parallel framework. In the following, we will conduct two ablation studies to discuss the contributions of different techniques in our proposed method.

4.3 Ablation Studies

Firstly, we conduct an ablation study on the recursive partitioning ternary tree (RPTT) in our framework by replacing the RPTT with the single recursive partitioning tree in ParaDRo [3]. The comparison results, shown in Tab. 3, reveal that the RPTT can reduce the runtime by 14% without obvious wirelength degradations.

[2]The FPGA24 contest focuses on runtime-first and non-timing-driven routing and requires the existing net routes to be preserved.

Table 3: The comparison between Ours w.o. RPTT and Ours. The ratios larger than 1 represent the quality degradation.

Benchmark	Runtime (s)	Wirelength	Score
logicnets_jscl	1.02	0.98	1.00
boom_med_pb	1.15	1.02	1.10
vtr_mcml	1.46	1.06	1.30
rosetta_fd	1.11	1.06	1.09
corundum_25g	1.03	0.76	0.95
finn_radioml	1.02	1.04	1.03
vtr_lu64peeng	1.12	1.02	1.06
corescore_500	1.08	1.01	1.04
corescore_500_pb	1.11	1.08	1.10
mlcad_d181_lefttwo3rds	1.16	1.11	1.15
koios_dla_like_large	1.14	1.04	1.12
boom_soc	1.42	0.98	1.32
ispd16_example2	1.01	0.99	1.00
Avg. Ratio	1.14	1.01	1.10

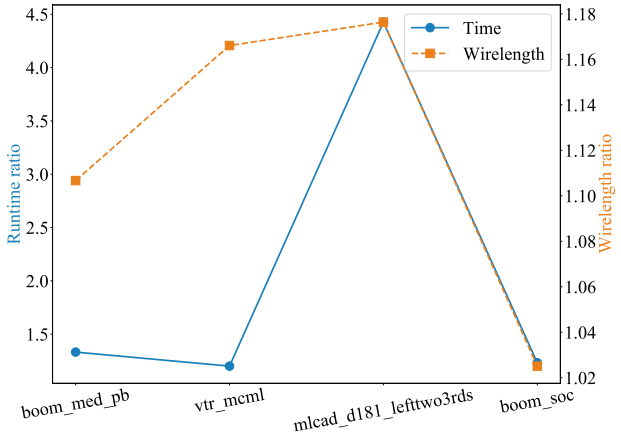


Figure 6: The comparison between Ours w.o HUS and Ours. The ratios larger than 1 represent degradation.

Secondly, we study the effect of the hybrid updating strategy (HUS) for congestion coefficients. We disable the HUS and apply the default updating strategy in RWRoute. The results on the four congested designs, depicted in Fig. 6, show that our HUS can both improve the runtime and the wirelength for congested designs. In particular, the runtime of mlcad_d181_lefttwo3rds is accelerated by around 4.5 times, and the wirelengths of mlcad_d181_lefttwo3rds and boom_med_pb are reduced by over 16%.

4.4 Impact of The Number of Threads

Furthermore, we also study the impact of the number of threads on our parallel router. As illustrated in Fig. 7, compared with the single thread, the runtime keeps reducing with the increase of thread number but will gradually converge at 32 threads.

5 CONCLUSION

In this work, we propose a fast parallel routing approach for commercial FPGAs, which incorporates a recursive partitioning ternary tree and a hybrid updating strategy for congestion coefficients. Compared with the commercial tool, our proposed approach has achieved significant improvements in both runtime and wirelength.

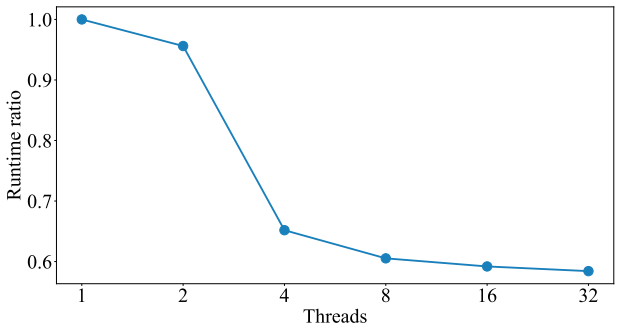


Figure 7: The average runtime ratio with different threads.

REFERENCES

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

[2] 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* 31, 1 (2011), 61–74.

[3] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 67–76.

[4] 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. [Online; accessed 27-Feburary-2024].

[5] Chris Lavin and Alireza Kaviani. 2018. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 133–140.

[6] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. *Third International ACM Symposium on Field-Programmable Gate Arrays* (1995), 111–117. https://api.semanticscholar.org/CorpusID:14212822

[7] Yehdhih Moctar, Mirjana Stojilović, and Philip Brisk. 2018. Deterministic parallel routing for FPGAs based on Galois parallel execution model. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 21–214.

[8] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. 2020. Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 2 (2020), 1–55.

[9] Minghua Shen and Guojie Luo. 2017. Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 105–114.

[10] Minghua Shen, Guojie Luo, and Nong Xiao. 2020. Combining static and dynamic load balance in parallel routing for fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (2020), 1850–1863.

[11] 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* 39, 4 (2019), 830–842.

[12] Yue Zha and Jing Li. 2022. Revisiting PathFinder Routing Algorithm. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Virtual Event USA, 24–34.

[13] 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.

[14] 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.