ACCELERATING THE PHYSICAL DESIGN OF LARGE FPGAS THROUGH DIVIDE-AND-CONQUER METHODOLOGY

Wanzheng Weng[1]*, and Pingqiang Zhou*[1*]
[1]School of Information Science and Technology, ShanghaiTech University, Shanghai, China
*Corresponding Author's Email: zhoupq@shanghaitech.edu.cn

2025 Conference of Science and Technology of Integrated Circuits (CSTIC) | 979-8-3315-1335-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/CSTIC64481.2025.11017777

ABSTRACT

The scale and complexity of circuit designs deployed on FPGA has surged with the increasing capacity of FPGA devices. At the same time, the runtime of physical design has grown exponentially, significantly extending the cycle of design iteration for engineers. To address this issue, we propose an automated, split-and-parallel physical design flow to accelerate the deployment of large-scale circuits on FPGA. We partition the original design into multiple sub-designs, perform placement and routing of each sub-design parallelly, and then merge them together. Experimental results show that our flow achieves 1.85X-2.7X speedup compared to standard Vivado flow with trivial degradation on design performance.

INTRODUCTION

The logic capacity of FPGA devices has expanded exponentially with the rapid advancements in semiconductor technology. The largest FPGA nowadays consists of more than 4 million lookup tables (LUT) and 8 million flipflops (FF). The growth in capacity has further boosted the deployment of increasingly larger and more complex digital systems on FPGAs. However, the ever-growing complexity of circuits presents significant challenges for physical design, particularly in terms of runtime. The time-consuming process of placement and routing significantly extends the cycle of design iteration and hinders the deployment of larger systems on FPGAs.

There have been dozens of works trying to accelerate the process of physical design for FPGAs, which mainly fall into two categories. The first kind of works try to parallelize the placement [1] or routing algorithms [2] and achieve acceleration through concurrent execution on multi-core CPUs or GPUs. However, most algorithms related with physical design are inherently sequential and the achievable parallelism are limited. Besides, most of these works don't take timing optimization into account, which is one of the most time-consuming processes in physical design.

The other kind of works attempt to partition the original design into multiple smaller parts, perform implementation, including placement and routing, of each part parallelly and then merge them together [3][4]. Following this divide-and-conquer paradigm, the work [4] can achieve 5-7X reduction in runtime of physical design compared to Vivado. However, all these approaches are exclusively applicable on HLS designs. The feasibility of these methods relies on the fact that HLS is written in untimed high-level languages, allowing the compiler to optimize the generated RTL codes for subsequent design partitioning. Consequently, these methods can't be extended to more general use cases where RTL codes are implemented manually.

In this work, we propose an automated, split and parallel physical implementation flow to speed up the deployment of large RTL designs on FPGAs. Similar to [3][4], we adopt the divide-and-conquer methodology by partitioning original design into multiple sub-designs, perform implementation of each sub-design concurrently, and then merge them into the complete design. In particular, the entire flow consists of three main stages as Figure 2 shows. In the first stage, we perform timing-path aware clustering on the synthesized netlist to generate an abstracted netlist. In the second stage, the entire FPGA region is divided into multiple islands; and nodes in the abstract netlist are placed into islands evenly. In the last stage, we perform physical implementation of each island parallelly using Vivado and then stitch all islands together to form the complete design.

We evaluate our split and parallel implementation flow on the Xilinx Ultrascale+ device using a set of large-scale benchmarks with varying architectural patterns. Experimental results show that our flow can achieve 1.8-2.5X speedup compared to the standard Vivado flow with only 1.8% degradation in design performance.

Challenges of Divide-and-Conquer Methodology

There are two critical issues arising with divide-and-conquer methodology that can significantly impact the quality of the merged design. The following section details these two issues and strategies we have applied.

The first issue is related with timing closure of inter-partition timing paths. For split implementation, a timing path may be partitioned into multiple segments and each segment is optimized independently during placement and routing. The invisibility of delay of other segments may lead to timing violation on the final merged path. An intuitive solution is to partition the design in such a way that cells on each timing path fall within the same partition. To achieve this, we propose a timing-path-aware clustering algorithm that encapsulates all combinational cells and their connected nets into virtual nodes so that they can be split in subsequent partitioning.

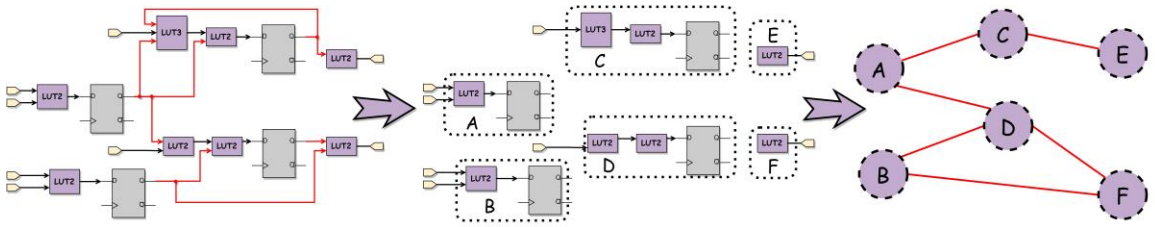

Figure 1: The procedure of timing-path-aware netlist clustering

The second issue concerns setting proper location constraints on cells connected with inter-partition nets. In the monolithic placement, cells connected by a net tempt to be dragged closer to minimize the total wirelength. In the split and parallel implementation, however, connected cells may span multiple partitions and are placed and routed independently. If no constraints are imposed on these boundary cells, they may end up being placed far apart, leading to extremely high delay and routing issues in the merged design. To address this issue, we adopt the idea of anchor registers from [4]. In detail, the source register of every inter-partition net is extracted out and assigned to the preserved boundary region between neighboring partitions. These registers act as anchors to guide the placement of boundary cells and achieve timing isolation between partitions.

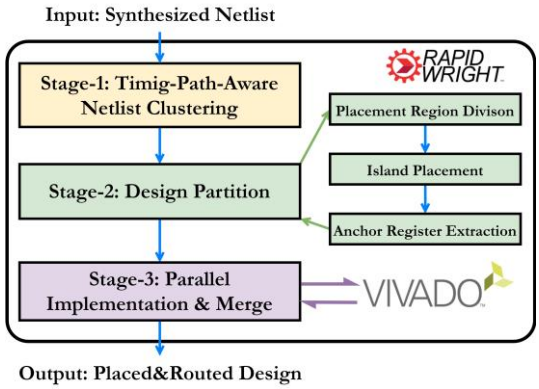

Figure 2: The proposed parallel implementation flow

METHODOLOGY

The entire proposed flow consists of three main phases as depicted in the Figure 2. The following sections provide detailed description of these steps respectively.

Timing-Path-Aware Clustering

As mentioned above, it's desirable that cells of one timing path are assigned together so that partitions are timing independent of each other. To achieve this goal, we develop a timing-path-aware clustering algorithm to generate abstracted netlist before design partition.

The procedure of the proposed algorithm is illustrated in Figure 1. The complete algorithm can be divided into three steps. In the first step, all nets driven by registers are removed from the synthesized netlist. Next, we search all connected components in the remaining hypergraph of the netlist and cluster them into new nodes. Finally, we traverse all removed edges, identify those spanning multiple clusters, and then connect new nodes to form the output abstract netlist.

Design Partition

Based on the abstracted netlist, the entire design is partitioned into multiple smaller sub-designs for parallel placement and routing. Partitioning a circuit design typically involves two key aspects: splitting the netlist into multiple parts and allocating disjoint and well-sized placement region to each part. We address this problem of floorplanning through the following three steps:

(a) **Division of Placement Region**: The entire placement region of the FPGA device is uniformly divided by grid, and each sub-region is referred to as an island. Additionally, a thin region is reserved between the boundaries of adjacent islands, which is referred to as anchor region. In this work, the entire device region is split using a 2x2 grid into four disjoint islands as Figure 3 shows.

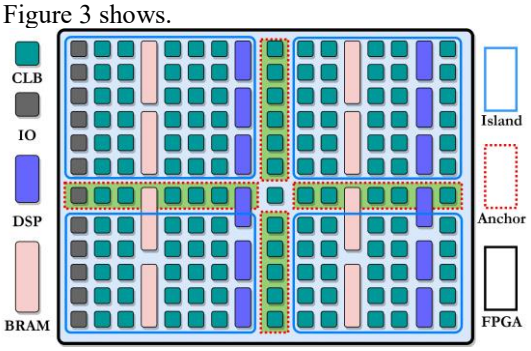

Figure 3: Division of the entire FPGA device

(b) **Island Placement**: After the division of placement region, cells in the abstracted netlist are placed to these islands such that the number of inter-island nets is minimized. Additionally, the island placement should satisfy the following two constraints: 1) the number of cells assigned to each island should be balanced; 2) each net can cross boundary at most one time. To solve this problem, we develop our solution based on min-cut placement algorithm [5]. This

algorithm determines the location of cells through a series of recursive min-cut bi-partitions. As shown in Figure 4, the entire placement region and netlist is first divided horizontally. Then each sub-netlist is further partitioned vertically into smaller sub-regions. Additionally, we need to set appropriate fix node constraint for the final bi-partition to ensure that every net only spans adjacent islands.

(c) **Anchor Register Extraction**: All edges in the abstract netlist are registered, and it's guaranteed that any edge can only span adjacent islands during island placement. Therefore, we can extract the source registers of cross-boundary nets and assign them to the corresponding anchor region to achieve timing isolation between islands and guide the placement of boundary cells.

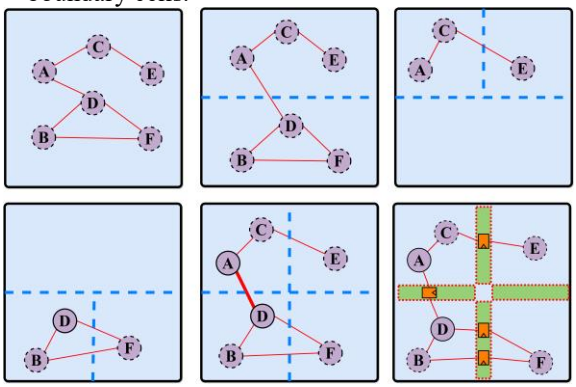

Figure 4: The entire flow of design partition

Parallel Physical Implementation and Merging

After partition, we perform parallel implementation of sub-designs and then stitch them together. Since anchor regions are shared by adjacent islands, parallel implementation may result in location mismatch of shared cells and conflictions on routing resources. To address this issue, we need to fix the placement of shared cells before parallel implementation. To get appropriate locations of these cells, we create a sub-netlist consisting of all anchor cells and their neighboring cells and then perform placement on this sub-netlist. After sub-designs are stitched together, additional re-routing of nets incident to anchor cells is required to resolve any routing conflictions.

RESULTS

A. Implementation Details

We implement our proposed flow in Java based on RapidWright platform [6], which empowers users to manipulate netlist and implementation produced by Vivado to construct customized flow for physical design of FPGA. Besides, we build our partition-based island placement using open-source hypergraph partitioner, TritonPart [7]. We evaluate our flow on Ubuntu 22.04 with AMD EPYC 7543 CPU (3.73GHz, 32 cores).

B. Benchmarks

In our experiments, we target the Xilinx Ultrascale+ vu3p FPGA, which consisting of 394K LUTs, 788K FFs, 2280 DSPs and 720 BRAMs. To evaluate our proposed flow, we collect four open-source large-scale circuit designs and change their parameters to generate seven benchmark circuits.

C. Results

Table I shows the comparation of runtime and achievable maximum frequency between the standard Vivado flow and our split-and-parallel flow across different benchmarks. To evaluate the impact of our flow on design performance, we repeatedly execute both our flow and Vivado flow with the period constraint decreasing in interval of 0.2ns to find the maximum achievable frequency for each benchmark. Results show that our flow introduces an average frequency loss of 1.8% across all benchmarks. For runtime comparation, the timing constraint of each benchmark is set based on the maximum frequency achieved by our flow. Under the same timing constraint, our flow achieves an average speedup of 2.3X compared to Vivado.

TABLE I. COMPARATION OF RUNTIME(S) AND MAX FREQUENCY(MHZ) BETWEEN VIVADO AND OUR FLOW

Name	Vivado		Our flow	
	RT	Freq	RT	Freq
blue-rdma	1904	285.7	1090	277.7
nvdla-small	1192	250.0	650	250.0
nvdla-med	1676	288.6	744	188.6
nvdla-large	2080	196.0	778	196.0
ntt-small	1340	555.5	624	555.5
ntt-large	3472	555.5	1342	526.3
corundum	1978	277.7	796	263.1
Ratio	2.3	1.07	1.0	1.0

REFERENCES

[1] R. S. Rajarathnam, et al. "DREAMPlaceFPGA: An Open-Source Analytical Placer for Large Scale Heterogeneous FPGAs using Deep-Learning Toolkit." *in Proceedings of ASP-DAC2022*, Taiwan, 2022, pp. 300-306.

[2] M. Shen, et al. "Exploring GPU-Accelerated Routing for FPGAs." *in IEEE Transactions on Parallel and Distributed Systems*, vol. 30, 2019, pp. 1331-1345.

[3] Y. Xiao, et al. "Fast Linking of Separately-Compiled FPGA Blocks without a NoC." in *Proceedings of ICFPT2020*, Maui, HI, USA, 2020, pp. 196-205.

[4] L. Guo, et al. "RapidStream: Parallel Physical Implementation of FPGA HLS Designs." *in Proceedings of FPGA2022*, New York, 2022, pp. 1-12.

[5] A. E. Caldwell, et al. "Can recursive bisection alone produce routable placements?" *in Proceedings of DAC2000*, 2000, pp. 477–482.

[6] C. Lavin, et al. "RapidWright: Enabling Custom Crafted Implementations for FPGAs." *In Proceedings of FCCM2018*, Boulder, CO, USA, 2018, pp. 133-140.

[7] I. Bustany, et al."An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design." *in Proceedings of ICCAD2023*, San Francisco, 2023, pp. 1-9.