INTEGRATION, the VLSI journal 106 (2026) 102532

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

RapidPnR: Accelerating the physical design for FPGAs via design-level parallelism☆

Wanzheng Weng *, Pingqiang Zhou *

School of Information Science and Technology, ShanghaiTech University, Shanghai, China

ARTICLE INFO

Keywords:
FPGA
Physical design
Divide-and-conquer
Parallelism
Acceleration

ABSTRACT

The runtime of physical design has become a critical issue for FPGA development as the scale and complexity of circuit designs surge with the increasing logic capacity of FPGA devices. The time-consuming process of physical design significantly extends the cycle of design iteration, which heavily impacts the efficiency of debugging and architecture optimization of circuit designs. To address this issue, this work proposes a generic, fully-automated and split-and-parallel physical design flow to accelerate the deployment of large-scale circuits on FPGAs. Specifically, our flow automatically partitions the synthesized netlist into multiple smaller pieces, performs parallel physical design of each piece, and then merges them into the complete design. Evaluated on a set of real circuit benchmarks, our flow reduces the runtime by more than 50% and ensures nearly the same design frequency compared to the physical design flow provided by the commercial tool Vivado.

1. Introduction

As the scale and complexity of designs continue to grow, the runtime issue of physical design on FPGAs has become increasingly significant [1,2]. Although the logic capacity of FPGAs keeps expanding exponentially with rapid advancements in semiconductor technology, the time-consuming process of physical design heavily impedes the deployment of larger circuits on them [2]. Typically, the development of FPGA systems is an iterative process, which involves multiple rounds of architecture design, RTL implementation, logic synthesis, physical design and system testing. Slow physical design processes significantly extend the cycle of design iteration, which affects the efficiency of debugging and performance optimization.

In recent years, numerous efforts have been made to accelerate the physical design for FPGAs, and the methodologies can generally be classified into two categories, algorithm-level parallelism and design-level parallelism. Algorithm-level parallelism first decomposes key physical design algorithms, such as placement [3–5], routing [6–9] and static timing analysis (STA) [10], into multiple independent tasks, then executes these tasks in parallel on GPUs [3,4,9] or multi-core CPUs [5,6]. Although the acceleration is enhanced by the increasing parallel computing capabilities of GPUs or CPUs, the sequential nature of most physical design algorithms limits further speedup. Besides, most of these works do not consider timing optimization, which accounts for most runtime of high-performance physical design and is difficult to parallelize [1].

In contrast, design-level parallelism directly partitions the original design into multiple smaller pieces, performs physical design of each piece concurrently, and finally merges them into a completely placed and routed design [1,2,11–13]. Xiao et al. [2,11–13] partitions an entire FPGA into multiple disjoint blocks connected with a lightweight Network on Chip (NoC) or direct wires. They then distribute the processing elements (PEs) of High-Level Synthesis (HLS) designs across these blocks and implement each block in parallel. Rapidstream [1] first partitions dataflow-based HLS designs into a mesh of disjoint islands, then insert pipelines into inter-island connections, and finally perform the physical design of each island concurrently.

The state-of-the-art design-level parallelism works only focus on HLS designs with the dataflow architecture [1,12]. They perform partitioning at the architectural level of designs, which relies on the module hierarchy information provided by the design language. Besides, they utilize the latency-insensitive nature of interconnections between dataflow PEs to merge the partitioned sub-designs together. Therefore these approaches are constrained to specific circuit architectures and frontend design languages. In this work, we explore the idea of design-level parallelism at the netlist level. We try to answer this question: can we directly partition the synthesized netlist and then perform physical design of each sub-netlist in parallel?

Two critical issues arise when adopting design-level parallelism at the netlist level. The first issue is how to ensure the timing quality of

☆ This work was supported by the Science and Technology Commission of Shanghai Municipality (STCSM) under Grant 24JD1402500.
* Corresponding authors.
E-mail addresses: wengwzh2022@shanghaitech.edu.cn (W. Weng), zhoupq@shanghaitech.edu.cn (P. Zhou).

https://doi.org/10.1016/j.vlsi.2025.102532
Received 23 May 2025; Received in revised form 29 July 2025; Accepted 27 August 2025
Available online 4 September 2025

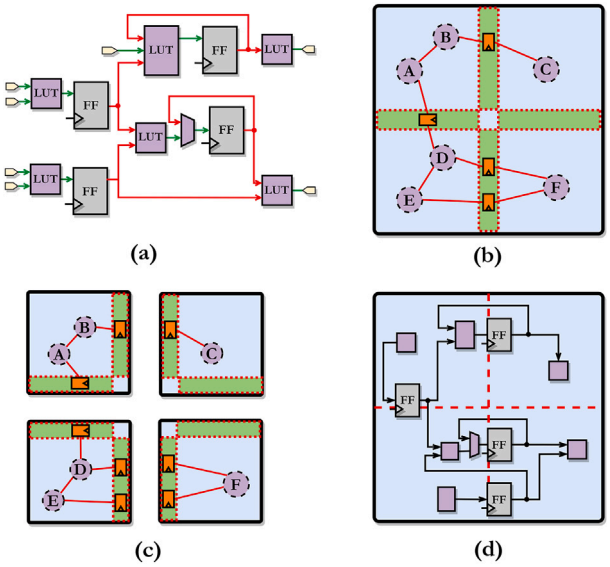

Fig. 1. An illustration of proposed split-and-parallel physical design flow: (a) the synthesized netlist produced by vendor tools; (b) transform the original netlist into an abstracted hypergraph, map this hypergraph into 2 × 2 islands, and extract and assign the source cell of each inter-island net to the boundary region; (c) split and parallel physical design of each island; (d) merge islands into the complete design.

inter-partition timing paths, while the second relates to routing and merging inter-partition nets, as well as the placement of cells incident to these nets. To address these two issues, we propose a fully automated, split-and-parallel and high-performance physical design flow to accelerate the placement and routing of the synthesized netlist. More specifically, our flow consists of three main stages, netlist abstraction, design partition and parallel placement and routing (PnR) & merging, as illustrated in Figs. 1 and 3. The first stage transforms the synthesized netlist into an abstract hypergraph. In this stage, we try to cluster combinational nets and cells and only expose registered nets in the abstract hypergraph to achieve timing isolation after design partition. In the second stage, the entire FPGA is divided into disjoint islands using a grid. Then we develop an efficient two-stage algorithm to distribute abstract nodes to these islands, ensuring multiple utilization constraints and minimizing the number of inter-island nets. To guide the placement and routing of inter-island nets, we adopt the idea of anchor registers [1] by extracting source cell of each inter-island net and assigning it to boundary region. Then we perform physical design of each island along with its neighboring anchor region in parallel and merge them into the complete design.

The main contributions of our work can be summarized as follows:

- To the best of our knowledge, our work is the first to explore the methodology of design-level parallelism at the netlist level to accelerate the physical design for large-scale FPGA designs.
- To ensure the legality and performance of the merged design, we propose a series of techniques, such as netlist abstraction, two-stage island placement and external delay prediction, and integrate them into an automated, split-and-parallel physical design flow which is independent of detailed circuit architectures or design languages.
- Evaluated on a set of diverse benchmark netlists, our flow speeds up the physical design by 1.6x–2.5x with an average frequency degradation of only 6% compared to the commercial tool Vivado [14].

2. Preliminaries

This section highlights key characteristics of the employed FPGA architecture that should be carefully considered in the physical design. Additionally, a brief introduction of FPGA physical design is also provided. In this work, our flow targets the AMD-Xilinx Ultrascale$^+$ FPGA [15], which is one of the most widely used FPGA families nowadays. A simplified Ultrascale$^+$ architecture is shown in Fig. 2.

2.1. Primitive organization

Under the Xilinx Ultrascale$^+$ architecture, the FPGA device primarily consists of seven types of primitive instances [16], including Block RAM (BRAM), Digital Signal Processing Unit (DSP), UltraRAM (URAM), Lookup Table (LUT), Carry Chain (CARRY), Multiplexer (MUXF) and Flip Flop (FF). These primitives are organized hierarchically within the device. At the lowest level of hierarchy, several LUTs, MUXFs, FFs and one CARRY are packed into a Configurable Logic Block (CLB) [17]. In addition to these primitive instances, each CLB site also contains site pins (which act as interface of CLB to externals), and site wires (which connect primitive instances and site pins). At the next level of hierarchy, sites of CLB, BRAM, DSP and URAM are arranged in columns within the device layout, with each column occupied by one type of primitive. Other hierarchical levels of the Ultrascale$^+$ architecture include the clock region (which consists of arrays of sites) and the super logic region (which arranges arrays of clock regions into a chiplet for multi-die FPGAs) [18].

2.2. Routing architecture

The interconnections between primitive instances in the Ultrascale$^+$ FPGA are established through two kinds of routing resources, including inter-site and intra-site.

Inter-site routing resources establish connections between different sites and can be further categorized into general and dedicated types. General inter-site routing resources connect pins of different sites through programmable switchboxes and metal wires of different lengths. As shown in Fig. 2(c), each site is directly connected with its neighboring switchbox and then this switchbox is connected with others through wire segments in four cardinal directions. Dedicated inter-site routing resources are hardwired connections between specific pins of adjacent sites in the same column [19]. These wires are usually used for the fast transmission of chaining signals between cascaded CARRYs, BRAMs or DSPs. For example, the COUT pin of lower CARRY and CIN pin of upper CARRY are connected directly through a wire without going through any switchboxes as shown in Fig. 2(d).

Intra-site routing resources consist of metal wires that connect primitive instances and site pins within a single CLB site. These wires can be categorized into local or global, depending on whether they are connected to site pins and can access inter-site routing resources outside the CLB site. For instance, as shown in Fig. 2(b), metal wires between CARRY and LUT, or MUXF and FF, are limited within one CLB site. While the input and output pins of LUTs or FFs can access inter-site wires through site pins to connect to FFs and LUTs in other CLB sites.

2.3. Physical design

Similar to the ASIC design flow, the deployment of RTL designs on FPGAs primarily involves two phases: synthesis and physical design. The synthesis process transforms RTL codes into a netlist consisting of primitive cells and nets that connects them. The physical design is responsible for implementing the synthesized netlist using primitive instances and routing resources available in the FPGA device. In detail, the physical design can be further divided into placement and routing. Placement maps cells in the netlist to primitive instances of the same

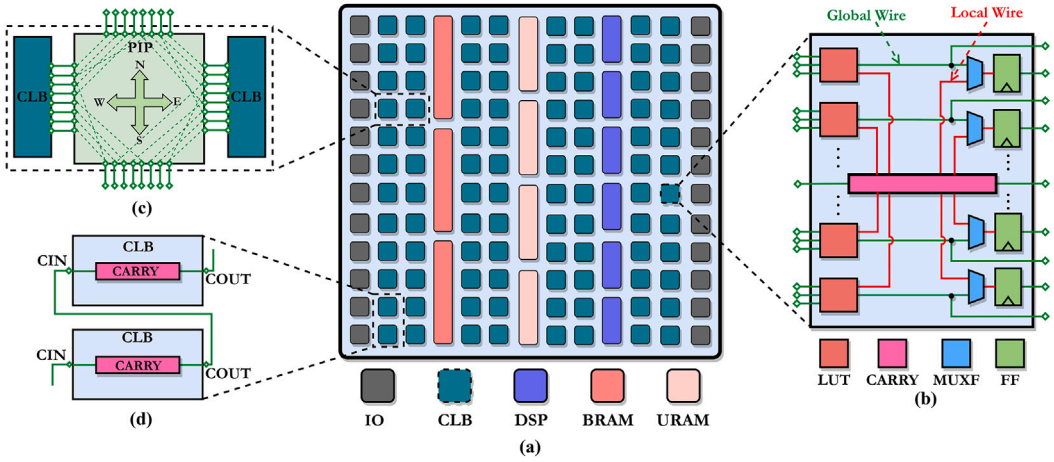

Fig. 2. A simplified architecture of AMD-Xilinx Ultrascale+ FPGA: (a) Primitive Organization: CLB, DSP, BRAM and URAM are arranged in a columnar way; (b) Internal structure of CLB site; (c) General inter-site routing resources consisting of PIP and wires connecting neighboring sites; (d) Dedicated inter-site wires connecting CIN and COUT ports of CARRYs in the vertically adjacent CLB.

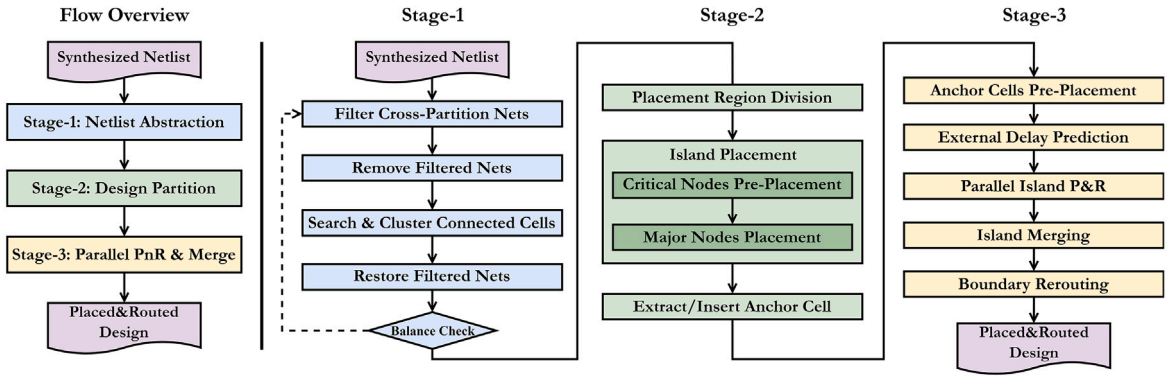

Fig. 3. The proposed split-and-parallel physical design flow for FPGAs.

type in the FPGA device while optimizing the total wirelength, timing quality and routability. Given locations of primitive cells, routing establishes the physical connections of nets in the netlist through programmable routing resources.

The characteristics of FPGA routing architecture impose certain constraints on design partition considering the legality of physical design. Specifically, it is undesirable that nets restricted to local intra-site wires or dedicated inter-site wires are cut during netlist partitioning. For example, if two cascaded CARRYs are partitioned into different columns in the device, nets of carry signals fail to be routed. Similarly, routing errors also occur if connected MUXF and FF are partitioned into different CLB sites.

3. The proposed split-and-parallel flow

The overview of the proposed split-and-parallel physical design flow is illustrated in Fig. 3. The entire flow mainly consists of three stages: netlist abstraction, design partition and parallel PnR & Merge. Netlist abstraction transforms the synthesized netlist into an abstract hypergraph with clustered primitives as nodes. During design partition stage, the abstract hypergraph is partitioned and floorplanned into disjoint regions within the device. After partitioning, each sub-design, consisting of both a sub-netlist and a constrained placement region, is placed and routed in parallel and then merged together into the complete design. The detailed description of each stage is provided in the following sections.

3.1. Netlist abstraction

The intuition behind netlist abstraction is to hide certain nets in the synthesized netlist through cell clustering. Actually it is aimed to impose restrictions on the later stage of design partition, considering both legality and design performance. For legality of the partitioned design, as discussed in Section 2.3, it is critical that nets restricted to dedicated inter-site or local intra-site wires remain uncut to avoid routing errors. With regard to design performance, it is desirable for all timing paths to reside within the same partition, thereby eliminating potential timing violations associated with inter-partition paths. To achieve this, we need to ensure that there are no cuts on any internal nets of each timing path. In other words, only nets driven by registers are allowed to be cut, which ensures that there is always a register at any junction of different partitions for timing isolation. The detailed implementation of netlist abstraction is described in Algorithm 1. And Fig. 4 shows the process of netlist abstraction on a simplified circuit netlist.

The first step of netlist abstraction is to filter out nets that should be concealed in the abstract hypergraph. For different designs, netlist abstraction adopts different filtering schemes. First of all, considering the legality of partitioned designs, nets restricted to dedicated and local routing resources are filtered out for any designs. Based on this fundamental criterion, any nets except for those driven by FF are also filtered out for timing isolation between partitioned islands. But experimental results in Section 4.3 show that clustering any connected combinational cells may produce extremely large vertices in the abstract hypergraph, which makes it impossible to get multiple balanced

Algorithm 1: Netlist Abstraction

Input: Original netlist hypergraph $G(E, V)$
Output: Abstracted netlist hypergraph $\hat{G}(\hat{E}, \hat{V})$

1 $isEdgeVisited(e) \leftarrow false$ **for** $e \in E$;
2 $removedEdges \leftarrow \{\}$;
3 $origin2AbstractMap \leftarrow \{\}$;
4 **foreach** *edge* $e \in E$ **do**
5 **if** *not isHiddenEdge(e)* **then**
6 isEdgeVisited(e) ← true;
7 removedEdges.add(e);
8 **foreach** *vertex* $v \in V$ **do**
9 **if** *origin2AbstractMap.contains(v)* **then**
10 continue;
11 queue ← $\{v\}$;
12 $\hat{v} \leftarrow \{v\}$;
13 origin2AbstractMap.put(v, $\hat{v}$);
14 **while** *not queue.isEmpty()* **do**
15 $v' \leftarrow$ queue.pop();
16 **foreach** *net e incident to* v' **do**
17 **if** *isNetVisited(e)* **then**
18 **continue**;
19 $isNetVisited(e) \leftarrow true$;
20 **foreach** *vertex* v'' *incident to e* **do**
21 $\hat{v} \leftarrow \hat{v} \cup v''$;
22 $origin2AbstractMap.put(v'', \hat{v})$;
23 queue.push(v'');
24 $\hat{V} \leftarrow \hat{V} \cup \hat{v}$;
25 **foreach** *net* $e \in removedEdges$ **do**
26 $\hat{e} \leftarrow \{\}$;
27 **foreach** *vertex v incident to e* **do**
28 $\hat{v} \leftarrow origin2AbstractMap.get(v)$;
29 $\hat{e} \leftarrow \hat{e} \cup \hat{v}$;
30 $\hat{E} \leftarrow \hat{E} \cup \hat{e}$;

sub-designs during design partition. So netlist abstraction process is performed iteratively by gradually relaxing net filtering criteria. If we cannot get balanced abstract hypergraph using aforementioned filtering criterion, we can relax it by allowing nets driven by LUT to be exposed to design partition.

In summary, a net is allowed to be exposed in the abstract hypergraph only if it satisfies the following two criteria: (1) not restricted to dedicated routing resources, and (2) driven by FF or LUT (optional). To identify whether a net satisfies certain criteria, we traverse all incident primitive cells of this net and see whether it connects specific ports of specific cells. For example, we can identify that a net is restricted to dedicated wires for carry signals if it connects the COUT and CIN ports of two cells of CARRY.

After filtering out nets to be concealed, we remove all other nets from the original netlist hypergraph. And then we search for connected components on this remaining hypergraph, corresponding to line 8–24 in Algorithm 1. Then all connected cells are clustered into a new node in the abstracted hypergraph. Finally, we traverse each previously removed net, identify all abstract nodes incident to it, and add an edge to connect these nodes in the abstract hypergraph.

3.2. Design partition

The stage of design partition is responsible for splitting the original netlist into multiple sub-netlists and allocate a disjoint and appropriately-sized region to each sub-netlist.

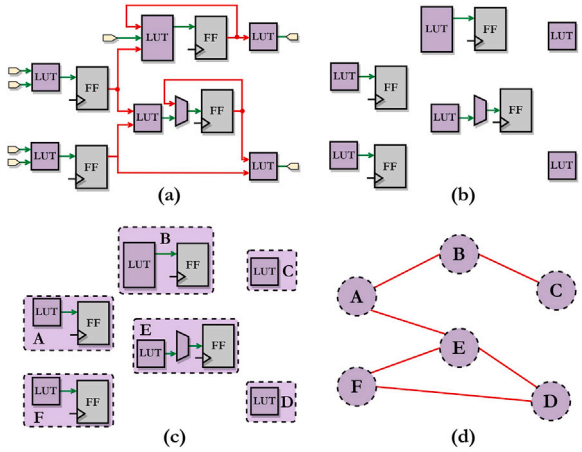

Fig. 4. The process of netlist abstraction on a simple netlist: (a) filtered netlist with hidden nets marked as green and removed nets marked as red; (b) remove all red nets from the netlist; (c) search connected cells in the remaining netlist and cluster them into abstract vertices; (d) retrieve previously removed nets and connect abstract vertices.

3.2.1. Placement region division

In the first step, the entire FPGA device is first divided into disjoint regions using a $M \times N$ grid, which we also refer to as islands. Additionally, a thin region is reserved along each boundary of adjacent islands, which is referred to as anchor region. These anchor regions only consist of several rows or columns of CLB sites. A simplified layout of the divided FPGA device is shown in Fig. 1(b), where anchor regions are highlighted in green.

3.2.2. Island placement

In the second step, vertices in the abstract hypergraph are distributed across this grid of islands. Specifically, island placement is aimed to balance the total size of vertices assigned to each island while minimizing the total number and length of inter-island nets. Besides it must guarantee that there is no over-utilization of each kind of resource on each island. There are two challenges related with island placement, considering the legality and efficiency respectively. For legality, island placement must simultaneously account for utilization constraints of multiple primitive resources, including BRAM, URAM, DSP, LUT, and FF. Additionally, the heterogeneity of each abstract vertex, which consists of different kinds of primitive cells, further increases the complexity of the problem. For efficiency, the number of abstract vertices varies from thousands to hundreds of thousands depending on different benchmark netlists, as shown in Section 4.3. Therefore, the time complexity of the adopted algorithm should be carefully considered to avoid significant runtime overhead on large-scale abstract hypergraphs. Considering these two challenges, we propose a two-stage algorithm to solve island placement, consisting of critical nodes pre-placement and major nodes placement.

The first stage only focuses on the placement of nodes containing critical primitive resources, including BRAM, URAM and DSP. Compared to LUT and FF, these resources are much more scarce and prone to result in over-utilization during partition. Due to their scarcity, the number of abstract nodes containing these resources is also relatively small, ranging from dozens to hundreds. Considering the scale of critical nodes, we can solve their placement optimally using Integer Linear Programming (ILP).

However, the placement of critical nodes may not be reliably optimized regardless of their interconnections with other nodes. To address this limitation, our pre-placement process incorporates not only critical nodes but also their strongly connected neighbors. These neighbors are

Table 1
Terminology and notation.

Notation	Description
M	The number of columns in the island grid
N	The number of rows in the island grid
R	Critical resources: BRAM, DSP and URAM
$G'(V', E')$	Partial hypergraph consisting of critical nodes and their neighbors
(x_v, y_v)	Location of node v in the island grid
I	Set of disjoint islands
(x_i, y_i)	Location of island i in the grid
$r(i)$	Amount Limit of resource $r \in R$ on island $i \in I$
$r(v)$	Amount of resource $r \in R$ in node $v \in V'$
α	Weighting factor of predicted net delay

selected based on their connectivity factor with critical nodes. Specifically, the connectivity factor of a node is defined as the total weight of edges incident to both this node and any critical nodes. A higher connectivity factor indicates stronger association with critical nodes. Using this metric, we construct a sub-netlist comprising critical nodes and their most strongly connected neighbors, then perform placement on this partial netlist to determine the location of critical nodes.

To formulate the ILP-based placement of this partial hypergraph $G'(V', E')$, we introduce the following variables, detailed descriptions of related notations can be found in Table 1.

- Coordinate of each node: (x_v, y_v) for each $v \in V'$
- Assignment of each node to each island: $a_{v,i} \in \{0, 1\}$ for each $v \in V'$ and each $i \in I$

Then the optimization goal is to minimize the total wirelength of partial hypergraph:

$$\sum_{e \in E'} \max_{v \in e} x_v - \min_{v \in e} x_v + \max_{v \in e} y_v - \min_{v \in e} y_v$$

And constraints are defined as follows:

- $0 \le x_v < M$ and $0 \le y_v < N$ for each $v \in V'$
- $a_{v,i} = 1$ iif $x_v = x_i$ and $y_v = y_i$
- $\sum_{i \in I} a_{v,i} = 1$ for each $v \in V'$
- $\sum_{v \in V'} a_{v,i} \times r(v) \le r(i)$ for each $i \in I$ and each $r \in R$

The first constraint ensure that the location of each node is not out of bound. The following two constraints build relation between location variable and assignment variable. The final constraint ensures that there are no over-utilization of each kind of resources.

After the location of critical nodes is determined, the second stage solves the placement of other major nodes in the abstract hypergraph. Considering the runtime efficiency of handling large-scale abstract hypergraph, we adopt the idea of min-cut VLSI placement algorithm [20, 21] to solve this problem. Specifically, the location of abstract nodes is determined through a series of recursive bi-partitions. Fig. 5 illustrates the process of distributing abstract nodes to a 2×2 grid through min-cut island placement. The abstract hypergraph is first partitioned in the dimension of y-axis and then the resulting two smaller partitions are further partitioned in the dimension of x-axis recursively. During the process of partitioning, we need to set appropriate assignment constraints for critical nodes to preserve the placement results from the previous stage. For each iteration of bi-partition, the latest multi-level hypergraph partition algorithm [22] is used to minimize cut size while balancing the size of two blocks.

However, this min-cut placement scheme only considers the optimization of total number of inter-island nets during partition. Therefore it may cause net crossing boundary more than one time, which may degrade the timing quality of related paths. For example, if nodes of certain net are distributed to diagonal islands, this net needs to cross two boundaries to connect incident nodes. Considering this problem,

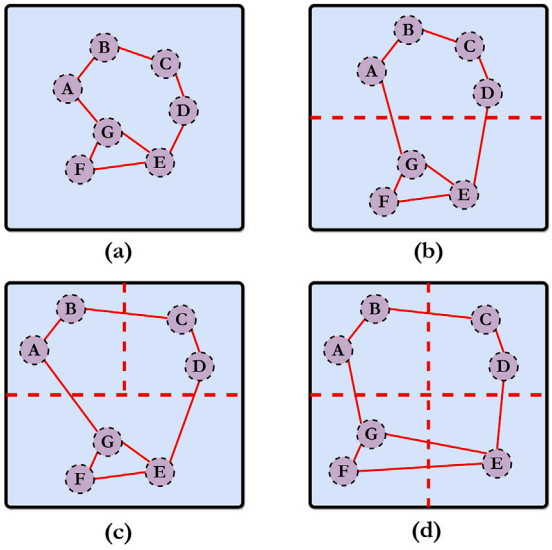

Fig. 5. An illustration of placement of major nodes through iterative bi-partition of the abstract hypergraph.

we also propose an enhanced version of min-cut island placement, which guarantees that each net only spans neighboring islands. To satisfy this constraint, we cluster any connected nodes that are incident to cut edges after each partition iteration, thereby hiding these nets and preventing them from being cut in the next iteration. The clustering of nodes incident to cut edges works similar to netlist abstraction introduced in Section 3.1. Specifically, we remove all edges except those that are cut during partition, search and cluster connected components in the remaining hypergraph, and then restore removed edges to connect clustered nodes.

Similar to netlist abstraction, clustering nodes incident to cut edges may result in extremely large nodes in the clustered hypergraph, which prevents balanced partition in the next iteration. Therefore strict single-boundary constraints on each edge may conflict with the goal of balanced island placement. In our flow, the actual island placement of major nodes is performed iteratively considering the trade-off between the length of inter-island edge and size balancing of each island.

3.2.3. Extract/insert anchor cell

After the location of each abstract node is determined, we extract the source cell of each inter-island net and assigned it to the corresponding anchor region, as shown in Fig. 1(b). Additionally, an extra buffer cell is inserted on nets crossing boundaries more than one time so that every crossed boundary has an anchor cell. Specifically, we use LUT1 primitive [16] as the buffer cell in the actual implementation. These anchor cells serve as the interface of neighboring islands and guide the placement and routing of connected cells.

3.3. Parallel PnR & merge

Before splitting and implementing each island, we first need to determine and fix the location of anchor cells shared by neighboring islands to ensure the consistency of their locations after parallel physical design. The optimal way to determine the location of anchor cells is to perform the placement of the entire netlist with region constraints on island and anchor cells. However, placing the entire design introduces significant runtime overhead, which conflicts with our goal of accelerating physical design by dividing the original large-scale problem into smaller, more manageable pieces. To reduce runtime overhead while obtaining appropriate locations of anchor cells, we perform placement on the partial netlist consisting of only anchor cells

and a fraction of island cells which are directly connected to them. The intuition behind this approach is that locations of anchor cells are primarily influenced by their neighboring cells.

As mentioned in Section 3.1, we attempt to conceal all combinational nets and expose only those driven by registers in the abstract hypergraph, in order to achieve timing isolation between partitioned islands. However, clustering all connected combinational cells may lead to excessively large vertices for certain benchmarks. In such cases, we have to expose some combinational nets to get balanced vertices, which may introduce inter-island timing paths after partitioning. To ensure the timing quality of the merged design, appropriate timing constraints must be applied to these paths before parallel implementation. In detail, for each island, we need to get the delay of the external segment of each inter-island timing path and set this value as the input/output delay on the corresponding ports. Since the actual delay of timing path segments is unknown before routing, we propose a simple and efficient prediction scheme to estimate its approximate value.

The prediction scheme works as follows: Firstly we model the delay of LUT as 1.0 and the delay of each net as $(1.0+\alpha \ln(fanout))$. Based on this simplified timing model, we adopt the basic static timing analysis algorithm to estimate the input and output delay of each pin in the netlist. Specifically, we first build a timing graph where each pin is modeled as a timing vertex and connection between each pair of pins is modeled as a timing edge. Then vertices in this graph are traversed in the topological order to calculate their input delay. For each vertex, we set its input delay as the maximum delay from all input edges. And similarly we traverses timing vertices in the reversed topological order again to calculate the output delay of each pin. Then we normalize these approximate input delay D_{in} and output delay D_{out} to actual clock period T_{clk} as follows:

$$D_{in} = \frac{D_{in}}{(D_{out} + D_{in})} \times T_{clk}$$

$$D_{out} = \frac{D_{out}}{(D_{out} + D_{in})} \times T_{clk}$$

After fixing anchor cells and setting constraints on inter-partition timing paths, we perform parallel physical design of split islands. When physical design of all islands are completed, we merge them into complete and fully placed and routed design. Because routing resources within anchor regions are shared by neighboring islands, there may exist routing conflictions on them when merging independently-routed islands together. So we perform extra rerouting of conflicted nets to ensure the integrity of merged designs.

4. Experiments

4.1. Implementation details

Our proposed flow is implemented in Java based on RapidWright platform [18,23], which provides a gateway to backend tools in the Xilinx FPGA development suite, Vivado. Using RapidWright, we are able to manipulate the synthesized netlist and physical implementation produced by Vivado to construct a customized physical design flow. All our experiments are conducted on a Linux server that consists of an AMD EPYC 7543 CPU @3.73 GHz (32 cores) and 125 GB memory. Some implementation details of each stage are outlined as follows:

4.1.1. Netlist abstraction

We parse the synthesized netlist produced by Vivado using RapidWright and extract the type of each primitive cell and their interconnection information. Based on the extracted netlist information, the netlist abstraction process is implemented in Java as described in the Algorithm 1. In our current implementation, all edges in the abstract hypergraph are assigned with an identical weight of 1.0.

Table 2
Statistics of benchmark netlists.

Name	Field	LUT/FF/BRAM/DSP/URAM	Total
blue-rdma	Network	119k/128k/125/0/0	248k
nvdla-1	DL	120k/113k/128/32/0	244k
nvdla-2	DL	143k/130k/128/52/0	288k
nvdla-3	DL	194k/158k/128/65/0	370k
nantucket-1	Crypto	93k/110k/0/576/16	207k
nantucket-2	Crypto	185k/221k/0/1152/32	417k
corundum	Network	74k/71k/238/0/45	149k
minimap-1	Genetics	185k/144k/10/280/0	342k
ispd-fpga02	Synthetic	100k/66k/100/100/0	166k
hardcaml-ntt	Crypto	88k/60k/0/1024/192	172k

4.1.2. Island placement

It is tricky and critical to set the appropriate dimension of island grid. A high-dimensional grid splits the design into smaller pieces, which shorten the runtime of implementing individual islands. However, it also introduces more inter-island nets, which may increase the runtime of boundary rerouting and degrade timing quality of the merged design. Considering these trade-offs, we empirically choose 2×2 island grid to partition the FPGA into four disjoint regions in our experiments. Our island placement algorithm can also be extended to support other different sizes of grid. We use the open-source ILP solver, Google OR-Tools [24], to solve the ILP problem formulated for critical nodes pre-placement. And the min-cut island placement is implemented based on the hypergraph partitioner TritonPart [22].

4.1.3. Parallel PnR & merge

The physical design of each island is performed using the commercial tool Vivado, and multiprocessing is employed to parallelize the implementation of multiple islands. The placement and routing results of each island produced by Vivado are read and merged together through RapidWright. Then we launch Vivado again to reroute conflicted nets within boundary regions.

4.2. Benchmarks

In our experiments, we target the Xilinx Virtex Ultrascale+ vu3p FPGA, which has been widely adopted in recent FPGA EDA contests [25] and academic research [8]. This device is the latest single-die FPGA, which consists of 394K LUTs, 788K FFs, 2280 DSPs, 720 BRAMs and 320 URAMs. To evaluate our flow, we collect a set of open-source circuit designs, including corundum [26], blue-rdma [27], nvdla [28], nantucket [29], minimap [30], ispd16-fpga02 [31] and hardcaml-ntt [32]. These designs contain both synthetic netlist and real circuit designs, targeting diverse applications, including networking, deep learning (DL), cryptography and genetics. On the basis of these designs, we change configuration parameters of some designs to produce a total of ten benchmark netlists with varying resource utilization. Specifically, all our netlists are synthesized under the out-of-context mode [33], which means that we do not consider the placement and routing of top-level IOs buffers during physical design. The application field and detailed resource utilization of each benchmark netlist are shown in Table 2.

4.3. Netlist abstraction

In the netlist abstraction stage, we aim to expose only the nets driven by registers while concealing all other combinational nets in the abstract hypergraph, in order to achieve timing isolation between partitioned islands. However, experimental results reveal that clustering all connected combinational cells may lead to excessively large vertices in the abstract hypergraph for certain benchmarks. To ensure balanced island partitioning, we have to leave some nets driven by combinational cells in the abstract hypergraph during net filtering. Specifically, we

Table 3
Evaluation of two net filtering scheme.

Name	FF Only		FF & LUT	
	#Node	Max node	#Node	Max node
blue-rdma	791	29.7k (11.98%)	70.1k	1.3k (0.53%)
nvdla-1	17.7k	29.1k (11.95%)	153.3k	3.9k (1.59%)
nvdla-2	18.0k	54.7k (18.98%)	181.0k	4.8k (1.65%)
nvdla-3	21.9k	46.0k (12.45%)	224.5k	3.9k (1.05%)
nantucket-1	44.8k	1.6k (0.76%)	61.0k	907 (0.44%)
nantucket-2	89.5k	1.6k (0.38%)	125.1k	903 (0.22%)
corundum	4.8k	16.8k (11.29%)	82.3k	1.5k (0.99%)
minimap	14.5k	223.2k (65.34%)	182.7k	23.8k (6.98%)
ispd-fpga02	13.1k	136.1k (81.88%)	156.8k	3.0k (1.79%)
hardcaml-ntt	4.4k	32.6k (18.92%)	52.6k	3.4k (1.97%)

adopt two net filtering schemes during netlist abstraction: (1) **FF Only**: expose only nets driven by Flip Flop (FF); (2) **FF & LUT**: expose nets driven by either FF or Lookup Table (LUT). Evaluation of these two schemes across all benchmarks is shown in Table 3, which reports total number of vertices in the abstract hypergraph and the size of the largest vertex. Specifically, the size of an abstract vertex is quantified by the number of cells clustered within it, as well as the ratio of this number to the total number of cells in the netlist. And this ratio is used to determine whether the abstract hypergraph is balanced. For each benchmark, we preferentially apply the first scheme. If it results in excessively large vertices that hinder balanced island partitioning, we switch to the second scheme.

Based on experimental results, we identify certain characteristics of benchmarks that can produce balanced abstract hypergraph under the first filtering scheme. From the perspective of netlist, these benchmarks exhibit a higher ratio of FF cells to the total number of cells. Additionally, they contain fewer high-fanout control signals, such as register enable signals, driven by combinational logic. From an architectural perspective, these benchmarks feature simple pipeline structures with minimal data interaction between different pipeline stages. Additionally, these benchmarks typically achieve higher working frequency than those that cannot produce balanced abstract hypergraphs.

4.4. Runtime comparison

To evaluate the efficiency of our proposed flow, the runtime of each benchmark is compared with the standard physical design flow provided by Vivado [14]. We do not include other baselines because no current works of accelerating physical design are capable of handling synthesized netlists directly. For works [1,11] also based on design-level parallelism, they are only applicable to HLS designs. For works [3,5,6] of algorithm-level parallelism, they only target one stage of physical design, either routing or placement, and most of them are only applicable to synthetic benchmarks [31] with limited kinds of primitive resources. The runtime of physical design is highly related with timing constraints. For fair comparisons, we set the same clock period constraint for both our flow and the standard Vivado flow, using the minimum value at which Vivado can achieve timing closure [34]. Specifically, for each benchmark, we perform physical design using Vivado repeatedly with the clock period decreasing in interval of 0.2 ns to find the tightest clock constraint. The clock period and runtime comparison of each benchmark are shown in Table 4.

The runtime breakdown of the overall flow and the final stage, parallel PnR & merge, is reported in Fig. 6 for benchmark **nvdla-3**. And similar runtime distributions are also observed on other benchmark netlists. The first two stages, including netlist abstraction and design partition, only account for about 2% of the total runtime. The majority of the runtime (about 97%) is consumed by the final stage, where we call Vivado to perform the physical design of islands and merge them together. In the final stage, the placement of anchor cells takes about 13% of the total runtime. And the remaining runtime of the last stage is

Table 4
Runtime comparison on all benchmarks.

Name	Period (ns)	Vivado (s)	Our flow (s)	Speedup
blue-rdma	3.6	1946	1195	1.63x
nvdla-1	5.2	1768	851	2.08x
nvdla-2	5.0	2204	1026	2.15x
nvdla-3	5.8	3244	1508	2.31x
nantucket-1	1.7	1458	801	1.82x
nantucket-2	1.8	3472	1570	2.21x
corundum	3.6	2316	912	2.54x
minimap	3.8	3462	1372	2.50x
ispd-fpga02	3.9	1970	928	2.12x
hardcaml-ntt	3.2	1580	1089	1.5x
Average	3.76	2342	1125	2.1x

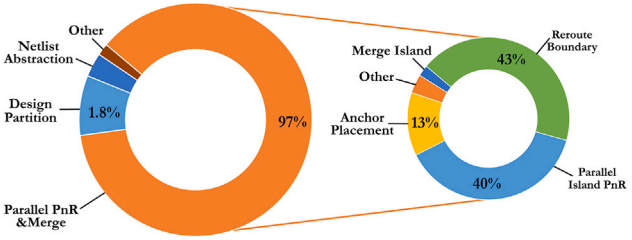

Fig. 6. Runtime breakdown of the overall flow and the stage of Parallel PnR & Merge on benchmark **nvdla-3**.

primarily consumed by rerouting the boundary region and the parallel placement and routing of islands, each accounting for about 40%. Noticeably, the step of rerouting boundary region induces significant runtime overhead. Even though only a few conflicted nets need to be rerouted, this step consumes nearly the same runtime as parallel placement and routing of each island, which heavily undermines the speedup achieved through our design-level parallelism scheme.

4.5. Frequency degradation

The impact of our split-and-parallel flow on design performance is also evaluated. We compare the maximum achievable frequency of our flow with that of the standard Vivado flow. The detailed comparison results are shown in Fig. 7. Our flow introduces an average frequency loss of 6% across all ten benchmarks. For most designs, our flow ensures no or only trivial frequency degradation (less than 0.2 ns increase in critical path delay). These designs all adopt the **FF Only** filtering scheme during netlist abstraction, which ensures that partitioned islands are timing-independent from each other. While there is relatively significant frequency degradation (about 0.8 ns increase in critical path delay) in several designs, such as ispd-fpga02 and hardcaml-ntt, which adopt the **FF & LUT** filtering scheme. This filtering scheme introduces inter-island timing paths, which impacts timing closure of the merged designs.

We further evaluate our proposed flow's impact on power consumption and total negative slack (TNS). The detailed comparison results between Vivado and our proposed flow are shown in Table 5. In terms of power consumption, our flow only introduces trivial increase on several benchmark netlists. The evaluation results of TNS demonstrates strong correlation with frequency or worst negative slack (WNS). The value of TNS is much higher on benchmarks with more significant frequency degradation.

4.6. Evaluation of proposed techniques

We further perform a comprehensive evaluation to validate the effectiveness of several proposed techniques, including netlist abstraction, anchor cell pre-placement and external delay prediction.

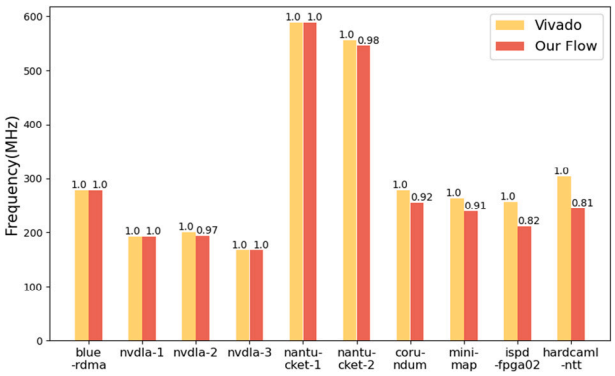

Fig. 7. Comparison of the achievable frequency between our flow and the standard Vivado flow.

Table 5
Comparison of power and TNS on all benchmarks.

Design	Vivado		Our flow	
	Power (W)	TNS (ns)	Power (W)	TNS (ns)
blue-rdma	5.1	0	5.233	0
nvdla-1	2.927	0	3.151	0
nvdla-2	3.17	0	3.056	−3.851
nvdla-3	4.817	0	4.444	0
nantucket-1	15.715	0	15.541	0
nantucket-2	29.421	0	29.718	−0.033
corundum	5.124	0	5.085	−0.318
minimap	2.937	0	2.768	−31.694
ispd-fpga02	20.457	0	22.651	−1213.958
hardcaml-ntt	15.252	0	15.721	−718.634

- **Netlist Abstraction**: We evaluate and compare the runtime and design frequency of two netlist abstraction schemes across several benchmarks. Detailed experimental results are presented in Table 6. The **basic** scheme only conceals critical nets, which may cause routing errors if their incident cells are split into different islands, to ensure the legality of the partitioned design. The **adaptive** scheme refers to our adopted netlist abstraction implementation, which considers both the legality and quality of the final designs. Experimental results demonstrate that the adaptive scheme improves the frequency of the final merged designs by 13% and reduces the runtime of the entire flow by 34.7%, highlighting its efficacy over the basic approach.

- **Anchor Cell Preplacement**: The impact of different anchor cell pre-placement strategies on the design performance is evaluated across four benchmarks, including **blue-rdma**, **nvdla-1**, **nantucket-1** and **ispd-fpga02**. Specifically, we compare the design frequency and runtime between random anchor pre-placement and our proposed pre-placement scheme introduced in Section 3.3. Experimental results demonstrate significant performance degradation with the random approach: a 27% reduction in design frequency and 90% increase in runtime across these benchmarks. Especially, **nantucket-1** exhibits severe routing congestion, with a 4.8× increase in the runtime; and **ispd-fpga02** suffers a 40% frequency degradation when using random pre-placement.
- **External Delay Prediction**: We evaluate the impact of external delay prediction on the final design performance using two benchmarks, including **minimap** and **ispd-fpga02**. The partition of these designs introduces inter-island timing paths, which requires proper timing constraints for parallel physical implementation. Experimental results reveal that without our delay prediction technique, the worst negative slack degrades by 0.06 ns for **minimap** and 0.94 ns for **ispd-fpga02**. On average, our approach improves the frequency of final merged designs by 9.3% across these two benchmarks.

Table 6
Effectiveness evaluation of netlist abstraction.

Design	Basic		Adaptive	
	Freq (MHz)	Runtime (s)	Freq (MHz)	Runtime (s)
blue-rdma	236.8	1365	277.78	1195
nvdla-1	182.95	2075	192.31	851
nvdla-2	177.71	1361	193.8	1026
nvdla-3	139.92	2489	172.41	1508
nantucket-1	514.67	835	588.23	801
nantucket-2	468.82	2966	545.55	1570
corundum	242.42	962	255.75	912
Average	280.47	1721.8	317.97	1123.3

5. Conclusion

In this paper, we explore the idea of design-level parallelism, originally applied to dataflow-based HLS designs, at the netlist level and propose a split-and-parallel FPGA physical design flow that is independent of specific circuit architectures or design languages. The proposed flow automatically partitions the synthesized netlist, places and routes each sub-netlist in parallel, and then merges them into the complete design. Experimental results show that our flow achieves an average speedup of 2.1x compared to the standard Vivado flow, with only about 6% degradation in design performance.

Although our proposed flow can achieve significant speedup with trivial timing degradation, there are still some limitations that need to be addressed in the future:

In terms of design quality, we observe non-negligible timing degradation in several benchmark circuits (e.g., **ispd-fpga02** and **hardcaml-ntt**). This degradation primarily stems from cross-island timing paths introduced during design partitioning, which adversely affect timing closure of the final merged design. Some potential methods to address this issue can be further explored in the future: (1) improve accuracy of external delay prediction to set more appropriate constraints on inter-island timing paths; (2) adopt timing-driven partition in min-cut island placement to reduce number of cuts on critical timing paths [22].

In terms of the efficiency, the bottleneck of the entire flow is boundary rerouting step in the final stage. The main reason is that the vendor tool follows the same flow as routing the entire design, in which many steps are time-consuming and unnecessary for rerouting conflicted nets. Rapidstream [1] develops a custom partial router based on the open-source router Rwroute [8] to speedup the process of rerouting conflicted nets, which may also be helpful for our flow. In addition to boundary rerouting, the runtime of parallel island placement and routing may be further optimized by dividing the design into smaller pieces using a finer grid.

In terms of the practicality, our current flow supports physical design only for single-die FPGAs. While modern multi-die FPGAs offer significantly greater resources (typically multiple times that of single-die devices) and enable the implementations of much larger circuit designs. And the physical design for these devices is more complex and time-consuming. Therefore, it is also worth extending our current flow to support multi-die FPGAs. To achieve this goal, we may need to consider the impact of high-latency inter-die routing interconnections on the timing quality of implemented design. Besides, the larger scale of benchmark netlist may also pose great challenges to the efficiency of our proposed partitioning algorithm.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Pingqiang Zhou reports financial support was provided by Science and Technology Commission of Shanghai Municipality (STCSM). If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, J. Cong, RapidStream: Parallel physical implementation of FPGA HLS designs, in: Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2022, pp. 1–12.

[2] D. Park, Y. Xiao, N. Magnezi, A. DeHon, Case for fast FPGA compilation using partial reconfiguration, in: 2018 28th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2018, pp. 235–2353.

[3] R.S. Rajarathnam, M.B. Alawieh, Z. Jiang, M. Iyer, D.Z. Pan, DREAMPlaceFPGA: An open-source analytical placer for large scale heterogeneous FPGAs using deep-learning toolkit, in: Proceedings of the 27th Asia and South Pacific Design Automation Conference, 2022, pp. 300–306.

[4] J. Mai, Y. Meng, Z. Di, Y. Lin, Multi-electrostatic FPGA placement considering SLICEL-SLICEM heterogeneity and clock feasibility, in: Proceedings of the 59th ACM/IEEE Design Automation Conference, 2022, pp. 649–654.

[5] W. Li, Y. Lin, D.Z. Pan, elfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs, in: 2019 IEEE/ACM International Conference on Computer-Aided Design, ICCAD, IEEE, 2019, pp. 1–8.

[6] T. Martin, D. Maarouf, G. Grewal, S. Areibi, A high-performance routing engine for large-scale FPGAs, in: 2024 34th International Conference on Field-Programmable Logic and Applications, FPL, IEEE, 2024, pp. 53–59.

[7] X. Zang, W. Lin, S. Lin, J. Liu, E.F. Young, An open-source fast parallel routing approach for commercial FPGAs, in: Proceedings of the Great Lakes Symposium on VLSI 2024, 2024, pp. 164–169.

[8] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, D. Stroobandt, RWRoute: An open-source timing-driven router for commercial FPGAs, ACM Trans. Reconfigurable Technol. Syst. (TRETS) 15 (1) (2021) 1–27.

[9] M. Shen, G. Luo, N. Xiao, Exploring GPU-accelerated routing for FPGAs, IEEE Trans. Parallel Distrib. Syst. 30 (6) (2018) 1331–1345.

[10] K.E. Murray, V. Betz, Tatum: Parallel timing analysis for faster design cycles and improved optimization, in: 2018 International Conference on Field-Programmable Technology, FPT, IEEE, 2018, pp. 110–117.

[11] D. Park, Y. Xiao, A. DeHon, Fast and flexible FPGA development using hierarchical partial reconfiguration, in: 2022 International Conference on Field-Programmable Technology, ICFPT, IEEE, 2022, pp. 1–10.

[12] Y. Xiao, S.T. Ahmed, A. DeHon, Fast linking of separately-compiled FPGA blocks without a NoC, in: 2020 International Conference on Field-Programmable Technology, ICFPT, IEEE, 2020, pp. 196–205.

[13] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, A. DeHon, Reducing FPGA compile time with separate compilation for FPGA building blocks, in: 2019 International Conference on Field-Programmable Technology, ICFPT, IEEE, 2019, pp. 153–161.

[14] AMD-Xilinx, Vivado ML edition 2023.2, 2023, https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.

[15] AMD-Xilinx, UltraScale architecture configuration user guide (UG570), 2024, https://docs.amd.com/v/u/en-US/ug570-ultrascale-configuration.

[16] AMD-Xilinx, UltraScale architecture libraries guide (UG974), 2024, https://docs.amd.com/r/en-US/ug949-vivado-design-methodology.

[17] AMD-Xilinx, UltraScale architecture configurable logic block user guide(UG574), 2017, https://docs.amd.com/v/u/en-US/ug574-ultrascale-clb.

[18] C. Lavin, A. Kaviani, Build your own domain-specific solutions with RapidWright: invited tutorial, in: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019, pp. 14–22.

[19] A. Samajdar, T. Garg, T. Krishna, N. Kapre, Scaling the cascades: Interconnect-aware FPGA implementation of machine learning problems, in: 2019 29th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2019, pp. 342–349.

[20] A.E. Caldwell, A.B. Kahng, I.L. Markov, Can recursive bisection alone produce routable placements? in: Proceedings of the 37th Annual Design Automation Conference, 2000, pp. 477–482.

[21] J.A. Roy, D.A. Papa, S.N. Adya, H.H. Chan, A.N. Ng, J.F. Lu, I.L. Markov, Capo: robust and scalable open-source min-cut floorplacer, in: Proceedings of the 2005 International Symposium on Physical Design, 2005, pp. 224–226.

[22] I. Bustany, G. Gasparyan, A.B. Kahng, I. Koutis, B. Pramanik, Z. Wang, An open-source constraints-driven general partitioning multi-tool for VLSI physical design, in: 2023 IEEE/ACM International Conference on Computer Aided Design, ICCAD, IEEE, 2023, pp. 1–9.

[23] C. Lavin, A. Kaviani, Rapidwright: Enabling custom crafted implementations for fpgas, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2018, pp. 133–140.

[24] Google, OR-Tools: an open-source, fast and portable software suite for solving combinatorial optimization problems, 2024, https://github.com/google/or-tools.

[25] I. Bustany, G. Gasparyan, A. Gupta, A.B. Kahng, M. Kalase, W. Li, B. Pramanik, The 2023 mlcad fpga macro placement benchmark design suite and contest results, in: 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD, MLCAD, IEEE, 2023, pp. 1–6.

[26] A. Forencich, A.C. Snoeren, G. Porter, G. Papen, Corundum: An open-source 100-gbps nic, in: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2020, pp. 38–46.

[27] Datenlord, Blue-rdma, 2024, https://github.com/datenlord/blue-rdma.

[28] Nvidia, Hardware manual of NVIDIA deep learning accelerator (NVDLA), 2017, https://github.com/nvdla/hw.

[29] Z. Contest, Nantucket, 2022, https://github.com/z-prize/2022-entries/tree/main/open-division/prize2-ntt/supranational.

[30] L. Guo, J. Lau, Z. Ruan, P. Wei, J. Cong, Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu, in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2019, pp. 127–135.

[31] I. Contest, ISPD 2016: Routability-driven FPGA placement contest, 2016, https://www.ispd.cc/contests/16/ispd2016_contest.html.

[32] Z. Contest, Hardcaml-ntt, 2022, https://github.com/z-prize/2022-entries/tree/main/open-division/prize2-ntt/hardcaml.

[33] AMD-Xilinx, UltraFast design methodology guide for Xilinx FPGAs and SoCs (UG949), 2024, https://docs.amd.com/r/en-US/ug949-vivado-design-methodology.

[34] Z. Xiong, R.S. Rajarathnam, D.Z. Pan, A data-driven, congestion-aware and open-source timing-driven FPGA placer accelerated by GPUs, in: 2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2024, pp. 115–125.