RapidStream 2.0: Automated Parallel Implementation of Latency–Insensitive FPGA Designs Through Partial Reconfiguration

LICHENG GUO, University of California Los Angeles, USA
PONGSTORN MAIDEE, AMD, Inc., USA
YUN ZHOU, Ghent University, Belgium
CHRIS LAVIN, EDDIE HUNG, and WUXI LI, AMD, Inc., USA
JASON LAU, WEIKANG QIAO, YUZE CHI, and LINGHAO SONG, University of California Los Angeles, USA
YUANLONG XIAO, University of Pennsylvania, USA
ALIREZA KAVIANI, AMD, Inc., USA
ZHIRU ZHANG, Cornell University, USA
JASON CONG, University of California Los Angeles, USA

Field-programmable gate arrays (FPGAs) require a much longer compilation cycle than conventional computing platforms such as CPUs. In this article, we shorten the overall compilation time by co-optimizing the HLS compilation (C-to-RTL) and the back-end physical implementation (RTL-to-bitstream). We propose a split compilation approach based on the pipelining flexibility at the HLS level, which allows us to partition designs for parallel placement and routing. We outline a number of technical challenges and address them by breaking the conventional boundaries between different stages of the traditional FPGA tool flow and reorganizing them to achieve a fast end-to-end compilation.

Our research produces RapidStream, a parallelized and physical-integrated compilation framework that takes in a latency-insensitive program in C/C++ and generates a fully placed and routed implementation. We present two approaches. The first approach (RapidStream 1.0) resolves inter-partition routing conflicts at the end when separate partitions are stitched together. When tested on the Xilinx U250 FPGA with a set of realistic HLS designs, RapidStream achieves a 5 to 7× reduction in compile time and up to 1.3× increase in frequency when compared with a commercial off-the-shelf toolchain. In addition, we provide preliminary results using a customized open-source router to reduce the compile time up to an order of magnitude in cases with lower performance requirements. The second approach (RapidStream 2.0) prevents routing conflicts using virtual pins. Testing on Xilinx U280 FPGA, we observed 5 to 7× compile time reduction and 1.3× frequency increase.

This work is partially supported by the CRISP Program, the CDSC Industrial Partnership Program, and the Xilinx Adaptive Compute Clusters Program.

Authors' addresses: L. Guo, J. Lau, W. Qiao, Y. Chi, L. Song, and J. Cong, University of California Los Angeles, 468A Engineering VI, UCLA, Los Angeles, CA 90095, USA; emails: {lcguo, lau}@cs.ucla.edu, wkqiao2015@ucla.edu, {chiyuze, linghaosong, cong}@cs.ucla.edu; P. Maidee, Y. Zhou, E. Hung, W. Li, and A. Kaviani, AMD, Inc., 2100 Logic Dr, San Jose, CA 95124, USA; emails: {pongstorn.maidee, eddie.hung, wuxi.li, alireza.kaviani}@amd.com, Yun.Zhou@ugent.be; C. Lavin, AMD, Inc., 3100 Logic Dr, Longmont, CO 80503; email: chris.lavin@amd.com; Y. Xiao, University of Pennsylvania, Levine Hall, 3330 Walnut St, Philadelphia, PA 19104, USA; email: ylxiao@seas.upenn.edu; Z. Zhang, Cornell University, Ithaca, New York, USA; email: zhiruz@cornell.edu.

1936-7406/2023/09-ART59 $15.00
https://doi.org/10.1145/3593025

CCS Concepts: • **Hardware → Hardware accelerators**; **Reconfigurable logic applications**; **Partitioning and floorplanning**; • **Computer systems organization → Dataflow architectures**; **High-level language architectures**; **Reconfigurable computing;**

Additional Key Words and Phrases: Multi-die FPGA, high-level synthesis, hardware acceleration, floorplanning, frequency optimization, HBM optimization

ACM Reference format:
Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2023. RapidStream 2.0: Automated Parallel Implementation of Latency–Insensitive FPGA Designs Through Partial Reconfiguration. *ACM Trans. Reconfig. Technol. Syst.* 16, 4, Article 59 (September 2023), 30 pages.
https://doi.org/10.1145/3593025

1 INTRODUCTION

Field-programmable gate array (FPGA) compilation techniques have traditionally been adopted from the electronic design automation (EDA) industry, where designers have a higher tolerance for a long turnaround time. However, this significantly impedes the adoption of FPGAs by the computing industry, where software programmers are used to a much shorter compile cycle [15, 41].

One general approach to speeding up FPGA compilation is to utilize multi-core CPUs or GPUs to parallelize the computer-aided design (CAD) algorithms, such as logic synthesis [20, 21], placement [1, 17, 22, 45–47], and routing [28, 29, 33, 60, 63, 66, 90]. However, many important algorithms used in the FPGA CAD toolflow are inherently sequential. Moreover, the slowest steps of the FPGA physical compilation extensively involve timing optimizations. Since optimizing timing typically requires global knowledge of the designs, it further increases the difficulty of parallelization. In Figure 1, we profile the CPU utilization of a 14-hour FPGA compilation task by the commercial Xilinx Vivado tool suite. As the figure shows, Vivado only uses 2.1 cores on average when attempting to close timing.

Another approach to fast FPGA compilation is to partition the application and then compile different parts in parallel. A new challenge naturally arises here — how to achieve timing closure with many inter-partition nets that connect different partitions? Given an RTL design or a netlist, it is relatively easy to partition the design and achieve timing closure within each partition, but it is difficult to achieve good timing on the inter-partition nets. Either we perform global cross-partition optimizations iteratively at the cost of high runtime overhead or we reduce the runtime at the cost of the timing quality of inter-partition nets.

Our prior work, RapidStream 1.0 [32], proposes an end-to-end split compilation flow for FPGAs that utilizes an architecture-level, latency-insensitive approach to address the timing closure challenges. Instead of targeting an arbitrary design, we focus on latency-insensitive designs where modules communicate through latency-insensitive protocols such as the AXI protocol or normal First In, First Out (FIFO). The motivating fact we have observed is that real-world large-scale designs are, in general, highly modularized and hierarchical, but existing CAD tools fail to utilize the architecture-level information of the input design. We instead propose that, if we partition the design only at the latency-insensitive boundaries, we can add extra pipelining to the boundary nets for timing closure without affecting the functionality of the design. Note that the proposed method requires "latency-insensitive" communication units but does not necessarily make the whole design latency-insensitive. In our previous work, AutoBridge [30], we have shown that this approach only adds about 10 to 20 clock cycles in the end-to-end latency for a set of convolutional neural network (CNN) accelerators and has less than 0.1% overhead on the total execution time.

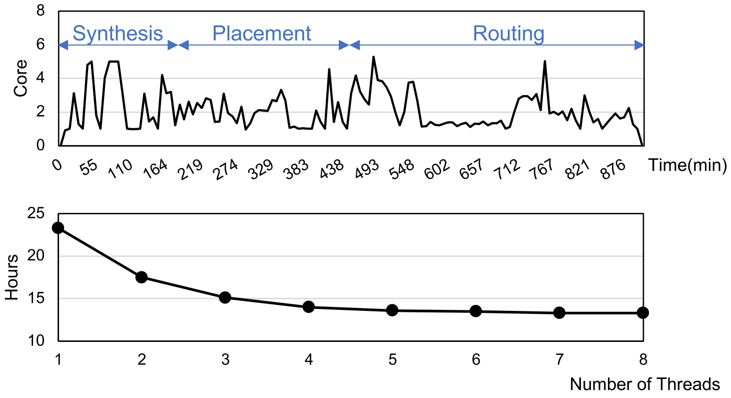


Fig. 1. The top figure shows the number of active CPU cores when implementing a CNN benchmark by Vivado (8 threads) on a 56-core server. The total implementation process takes about 14 hours, with an average CPU utilization of 2.1 cores. The bottom figure displays the runtime as we increase the number of threads.

As illustrated in Figure 3, RapidStream 1.0 includes three major phases. During the partitioning phase, we organize the FPGA device as a mesh of disjoint *islands* and floorplan a latency-insensitive design into the islands. We then utilize the pipeline flexibility to insert pipeline registers into the inter-island nets, which we call *anchor registers*. The anchor registers provide crucial timing isolation between islands to enable parallel implementation. Finally, we stitch together the layout results of each island to generate the complete implementation. The key technical contributions of RapidStream 1.0 are summarized as follows:

- To the best of our knowledge, we are the first to propose an automated, parallelized, and physically integrated flow to map a latency-insensitive design into a fully placed and routed FPGA implementation while achieving fast timing closure.
- We identify and address several technical challenges for a practical split compilation flow. Specifically, we propose new and effective methods for (1) inserting pipeline registers and optimizing their placement at the latency-tolerant borders of partitions, (2) clock management in parallel routing, and (3) efficient island stitching and routing of inter-island nets.
- Our evaluation shows that the proposed approach significantly increases the degree of parallelism of FPGA-targeted split compilation. RapidStream uses ~26 cores on average, whereas a commercial CAD tool only utilizes about two cores on average. As a result, we achieve an end-to-end speedup of 5 to 7× over the commercial tool. Additionally, we achieve an improvement in frequency by up to 1.3×.

The major limitation of RapidStream 1.0 is that it involves a global routing step after assembling the islands together because we need to address the routing conflicts between islands. Since a general-purpose router requires a lengthy initialization process, nearly half of the total compilation time is consumed by this step, even though only 5% of the nets need minor adjustments. To address this issue, in RapidStream 1.0, we proposed to adopt an open-source router, RWRoute, that specializes in quick initialization and fixing local routing conflicts. While RWRoute can finish the routing task quickly and efficiently compared with Vivado, the open-source router lacks an accurate hold time model and final results contain some hold violations. Therefore, in RapidStream 1.0, we still rely on the Vivado router to produce a fully legal solution and bear the overhead in router initialization.

In this extension article, we present RapidStream 2.0, which adopts a different approach to address the bottleneck in the routing step. Compared with our prior methods, we propose to first partially route the inter-island nets. The partial routes will connect the anchor register to a virtual pin inside the island. Although partial routing of anchor nets will also be performed globally, we propose methods to speed up the process by using a skeleton design instead of the full design. In our skeleton design, the majority of the inner-island elements are pruned away and only source/sink connections annotated with virtual pins to the anchor registers are preserved. By connecting to virtual pins, the global routing problem is decomposed into a local one and allows us to safely route each island independently. Since the virtual pins of boundary nets are all placed inside the island, we eliminate conflicts outside the island region. In this way, we do not need a separate global routing step at the end and can generate the bitstream of each island directly. These partial bitstreams will make a whole design when they are loaded onto the device.

Another major improvement in RapidStream 2.0 is that we have supported partial implementation, where part of the design is allowed to be pre-placed and pre-routed in an offline process. One notable application of this new feature is the integration with the AMD/Xilinx Vitis framework. The Vitis framework provides an efficient way to set up host–device communication. To do so, the Vitis framework provides a fixed *shell*, consisting of a group of pre-built IPs, including DMA, PCIe, DDR, HBM subsystem, and so on. In a Vitis development flow, the shell is pre-implemented in the boundary area of the chip and the user logic can use the remaining unoccupied area of the FPGA. By integrating support for pre-built shells, RapidStream 2.0 can support CPU-FPGA communication through Vitis.

We build a prototype of RapidStream 2.0 with the AMD/Xilinx Alveo U280 HBM boards and achieve 5 to 7X end-to-end speedup compared with a normal implementation process. Notably, our generated bitstream is fully functional and has passed onboard tests. Compared with RapidStream 1.0, we observe as much as 2× speedup in RapidStream 2.0.

This article is organized as follows. We first present the background information and the overview of our workflow in Section 2. Then, we present the major steps that are shared between versions 1.0 and 2.0: design partitioning (Section 3), parallel placement (Section 4), and clock management (Section 5). Next, we present the routing solutions of RapidStream 1.0 and 2.0. For RapidStream 1.0, we discuss island stitching and inter-island routing (Section 6). We present our efforts to further accelerate the inter-island routing step for RapidStream 1.0 (Section 7). Next, for RapidStream 2.0, we present how to set up the partial reconfiguration environment to enable parallel routing (Section 8). We also show our work-in-progress to speed up the construction of a partial reconfiguration environment through RWRoute. As for implementation, we first show experiment results of RapidStream 1.0 in Section 10, and then we evaluate RapidStream 2.0 in Section 11. We also compare our work with related works in Section 12.

2 PRELIMINARIES

2.1 Problem Scope

RapidStream focuses on latency-insensitive FPGA designs. By our definition, a latency-insensitive design consists of (1) a collection of *processing elements (PEs)* working in parallel and (2) a set of FIFOs that connect the communicating PEs. Each PE can be arbitrarily complex internally, but it must send or receive data through FIFO interfaces.

2.2 Organization of the FPGA Fabric

To facilitate the split compilation, we divide the FPGA fabric into two types of regions. As illustrated in Figure 5, these regions include (1) large disjoint *islands* (in blue) that are approximately

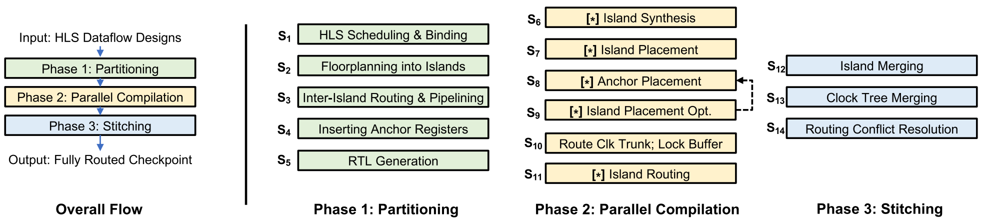

Input: HLS Dataflow Designs
Phase 1: Partitioning
Phase 2: Parallel Compilation
Phase 3: Stitching
Output: Fully Routed Checkpoint
Overall Flow
S1 HLS Scheduling & Binding
S2 Floorplanning into Islands
S3 Inter-Island Routing & Pipelining
S4 Inserting Anchor Registers
S5 RTL Generation
Phase 1: Partitioning
S6 [*] Island Synthesis
S7 [*] Island Placement
S8 [*] Anchor Placement
S9 [*] Island Placement Opt.
S10 Route Clk Trunk; Lock Buffer
S11 [*] Island Routing
Phase 2: Parallel Compilation
S12 Island Merging
S13 Clock Tree Merging
S14 Routing Conflict Resolution
Phase 3: Stitching

Fig. 2. An overview of our RapidStream workflow. We use [*] to denote a parallelized step.

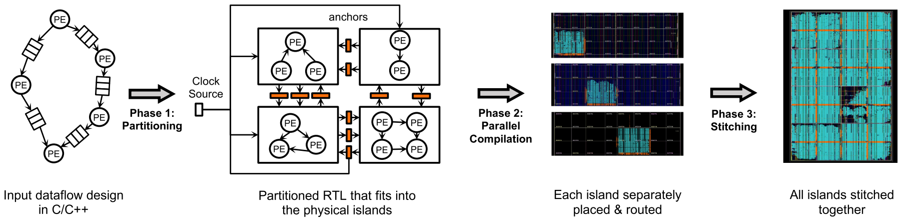

PE
Phase 1: Partitioning
Clock Source
anchors
Phase 2: Parallel Compilation
Phase 3: Stitching
Input dataflow design in C/C++
Partitioned RTL that fits into the physical islands
Each island separately placed & routed
All islands stitched together

Fig. 3. Illustration of results obtained in different phases. In the final output, the orange part shows the anchor registers and the cyan part shows the implemented partitions.

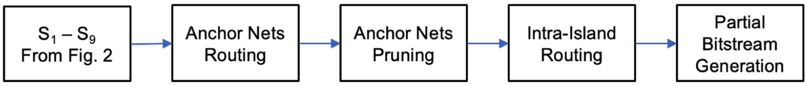


Fig. 4. Flow of RapidStream 2.0.

equal-sized and (2) thin columns/rows of *anchor regions* (in green) between adjacent islands. Here, we define an island as a square-shaped region reserved for (a subset of) the user logic; we further require that different islands are non-overlapping. The anchor regions are reserved to place the anchor registers (in orange) needed for inter-island communications; each inter-island connection is equipped with one anchor register, which isolates the inter-island timing paths.

Note that we need to distinguish the anchor regions located at die boundaries. The Xilinx multi-die FPGAs have discrete channels for die-crossing signals. To facilitate timing closure, the anchor registers will be placed in the die-crossing channels to bridge the islands that are on different sides of the die boundary (see Figure 5).

2.3 Flow Overview

Figure 3 shows the input and output of each phase of our proposed workflow. In Phase 1, we take in an HLS dataflow design and floorplan it to the disjoint islands (steps S_1 and S_2 in Figure 2). We take advantage of the elasticity of dataflow designs to ensure that every inter-island connection is pipelined with an *anchor register* (S_3 and S_4). This provides timing isolation that is crucial in the later parallel placement and routing.

Phase 2 performs parallel placement and routing of the disjoint islands and inserts the anchor registers. In the placement step (S_7–S_9), we propose to iteratively co-optimize the placement of anchors and islands since they are interdependent. In the routing step (S_{10}–S_{11}), we propose a clock management scheme to ensure that the clock skew is consistent when the islands are routed and later stitched together. Without this step, we will run into hold violations after stitching.

In Phase 3, we implement a stitcher using the RapidWright framework [42] to stitch the physical netlists of post-routing islands together (S_{12}, S_{13}). Although the nets inside each island remain legal

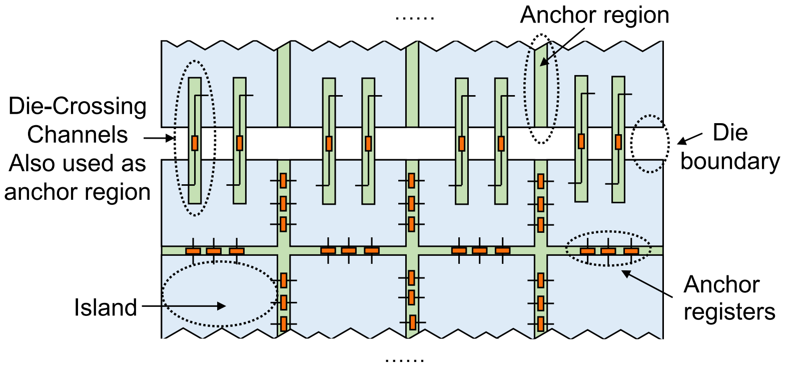


Fig. 5. Organization of the FPGA device.

after stitching, conflicts may arise among the inter-island anchor nets. This is a routing problem unique to our flow, and we propose a lightweight method to resolve the potential routing conflicts (S_{14}). Compared with the full-fledged commercial router, we achieve a 4× speedup on average while retaining nearly the same setup slacks.

3 PARTITIONING

This section describes steps S_1 to S_5 of the partitioning phase of RapidStream, as shown in Figure 2.

3.1 Problem Description

In this phase, we exploit the pipelining flexibility of HLS to transform the design into a parallelization-friendly structure. We first discuss what features are needed in later phases that parallelize the physical implementation of islands.

Objective 1: Non-overlapping partitioning – Since we aim to parallelize the physical implementations of different islands, each island is required to host a unique and non-overlapping partition of the original design.

Objective 2: Pipelined inter-island connections – To facilitate the timing closure on the inter-island nets, we want each inter-island connection to be pipelined with an *anchor* register.

Objective 3: Direct neighbor connections – We further enforce that each island only has direct connections with adjacent islands. This property is key to parallelizing the placement and routing process.

3.2 Approaches

Next, we introduce how RapidStream partitions and transforms the original dataflow design to satisfy the above-mentioned objectives.

Mapping PEs to Islands (S_2). To achieve objective 1, we exclusively assign each PE to one island. The assignment problem is formulated as follows:

The input dataflow design is represented as a graph $G(V, E)$, where each vertex $v \in V$ represents one PE; each edge $e_{ij} \in E$ represents an inter-PE FIFO connection between v_i and v_j. Given an array of islands that has N rows and M columns, the goal is to map each $v \in V$ to one unique island such that the resource of each island is not overused and the total wirelength is minimized. We use the weighted Manhattan distance to calculate the total wirelength:

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|), \tag{1}$$

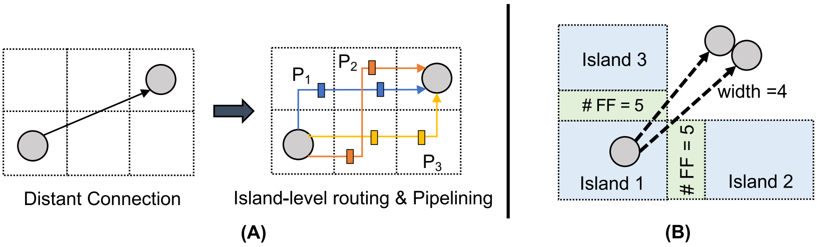


Fig. 6. (A) Three potential routes for a connection. (B) Each anchor region (in green) only has 5 Flip-Flops, so that the two connections (both of width 4) cannot go through the same anchor region.

where $e_{ij}.width$ is the bitwidth of the FIFO between v_i and v_j and each v is assigned to the $v.col$-th column and the $v.row$-th row.

The rationale behind the formulation is that a shorter wirelength results in a lower latency overhead. Our problem is typically small in size since an HLS design usually only instantiates up to a few thousand PEs. Hence, we use integer linear programming (ILP) to formulate and solve a top-down partitioning-based placement problem iteratively. Notably, the placement problem is similar to the ones described in several prior works [2, 24, 30, 50].

Global Planing and Pipelining Inter-island Connections (S_3). Before we pipeline the connections between non-adjacent islands, we need to first determine which intermediate islands the connections will go through. Essentially, we need to first solve a routing problem at the island level. Next, we insert pipeline registers in the islands that the connection passes through. As an example, Figure 6(A) shows the potential routes (P_1, P_2, P_3) for a connection between two non-adjacent islands.

The main constraint in this routing problem is the number of available flip-flops (FFs) in the anchor regions. Recall in Figure 5 that we reserve a thin region between islands to hold the anchor registers for inter-island nets and each inter-island net has an anchor register. Therefore, when routing the connections at the island level, we must ensure that the participating anchor regions have sufficient FFs for pipelining all the nets passing through, as illustrated in Figure 6(B).

Since the number of islands being mapped to is typically small, we again formulate the problem in ILP. For each connection, we generate all potential routes with the shortest Manhattan distance that have at most two bends. For each anchor region between a pair of adjacent islands, we add a constraint to ensure that the number of passing-through nets is no greater than the available FFs. We also assign a cost to each route based on the average resource utilization of the passing islands. The ILP is set up to minimize the total cost in this path selection problem.

Inserting Anchor Registers (S_4). To facilitate timing closure and inter-island routing, each island will register all input/output signals. Figure 7 shows how we insert anchor registers into the inter-island nets between adjacent islands. We leverage an *almost-full FIFO*, which asserts the `full` signal before the FIFO is actually full. This signal increases tolerance of the round-trip latency between adjacent islands, which allows us to add a pipeline register without causing an overflow.

Note that we choose to use the ILP formulations because they are sufficiently fast and scalable for today's HLS designs and FPGA devices. This is validated by our experiments in Section 10. For future FPGA designs that may become much larger, we can incorporate other well-known techniques, such as multi-level placement [6] and hierarchical routing [78], to handle the increased complexity.

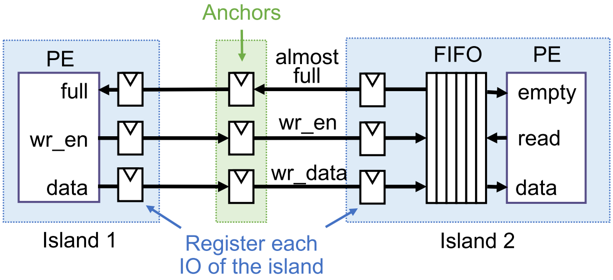


Fig. 7. Inserting anchor registers.

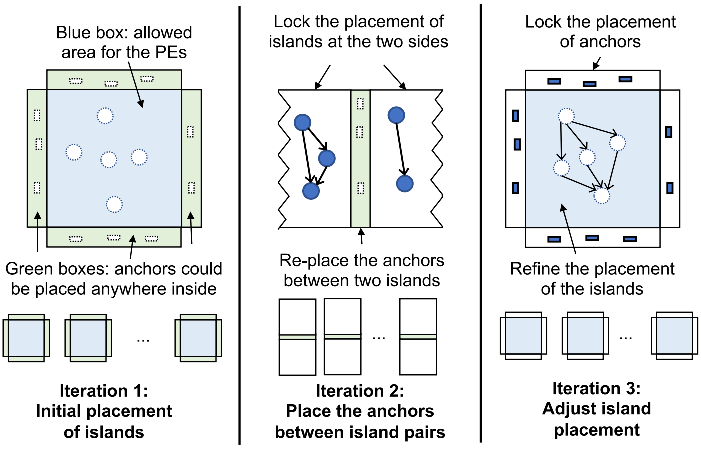

Blue box: allowed area for the PEs
Green boxes: anchors could be placed anywhere inside
Iteration 1: Initial placement of islands
Lock the placement of islands at the two sides
Re-place the anchors between two islands
Iteration 2: Place the anchors between island pairs
Lock the placement of anchors
Refine the placement of the islands
Iteration 3: Adjust island placement

Fig. 8. Demonstration of the iterative placement.

4 PARALLEL PLACEMENT

Phase 1 produces an optimized version of the RTL that is floorplanned to the island regions and anchor regions (see Figure 5). In step S_2, we determine which PEs are assigned to each island region; in step S_3, we compute which anchor registers each anchor region accommodates. In Phase 2, we first synthesize the RTL of each island into the netlist representation (S_6). As all islands are non-overlapping, we are able to run logic synthesis for all islands in parallel. Next, we place all island regions and anchor regions in parallel based on the previous floorplanning (S_7–S_9).

4.1 Iterative Placement of Anchors and Islands

Compared with logic synthesis, it is more challenging to parallelize the placement step. Two neighbor islands that are independently placed should have their interface properly aligned. This requires the separate placer processes to properly synchronize on inter-island connections.

We adopt an iterative approach to gradually align the interfaces of separately placed islands by utilizing the anchor regions between islands. Figure 8 sketches the main ideas of our approach. The intuition is that we lock the placement of all islands and then incrementally re-place the anchor regions, then alternate their roles in the next iteration.

Iteration 1 (S_7). In the first iteration, we determine an initial placement of the islands. To place an island by itself, the placer needs the locations of all anchors around the island, which are unknown at the time. Thus, we only impose a partial constraint that each anchor should be within the anchor region on its corresponding side of the island.

Iteration 2 (S_8). With the initial placement of each island, we compute the exact locations of the anchors between the islands to connect the inter-island nets. This step is also carried out by parallel placer processes. Each process handles a pair of adjacent islands and places the anchors in between to best connect both sides. We further elaborate this step in Section 4.2.

Iteration 3 (S_9). We fine-tune the placement of islands based on the exact anchor locations. Since the resulting anchor locations from the first two iterations may differ, iteration 3 further refines the placements of the islands to best match the latest anchor locations from iteration 2.

Through the three iterations, all islands are placed in a parallel manner. It is possible to repeat iteration 2 (S_8) and iteration 3 (S_9) to further improve the overall timing quality. However, our experiments indicate that applying them just once is enough to achieve around 400 MHz based on post-placement estimation.

4.2 Anchor Placement by Min-Cost Matching

Motivation. While we use the standard placer for iterations 1 and 3, we formulate the anchor placement problem (iteration 2) as a min-cost matching problem. Iteration 2 places the anchors based on the placement of the islands on the two sides. First, since the anchor region is very thin,[1] it is effectively a 1-D placement problem and the solution space tends to be small. Second, using the standard placer would incur unnecessary overhead in compile time, as it is optimized for general situations. Finally, we need control in a finer granularity to make sure that all anchors are exactly inside the feasible regions.

Method. We propose a simple yet effective distance-driven placement formulation specifically for iteration 2 (S_8) that can achieve a similar timing quality compared with a standard placer but with a much shorter runtime. Given an anchor, we assign a heuristic value for each FF in the anchor region representing the cost to place the anchor onto that FF. Then, we minimize the total cost of placing all anchors. This formulation is a min-cost matching problem that can be solved in polynomial time [7]. Specifically, we formulate the problem in linear programming (LP), which in this case guarantees integer solutions because the constraint matrix is totally unimodular [37].

We use a heuristic method to determine the cost function. To place an anchor onto an FF, the cost consists of two parts: (1) the total wirelength from the anchor to the source and sink cells; and (2) the wirelength difference between the longest and the shortest net of the anchor. We sum the two parts with empirical weights. This distance-based heuristic will push the anchors close to their source and sink cells and avoid being too close to one cell but far away from the other.

Consider the example in Figure 9, where we need to place two anchors to four potential FFs (A, B, C, and D) between the islands. Since the source and sink of anchor 1 are at the top, A has a smaller cost than the others. Likewise, D has the smallest cost for anchor 2.

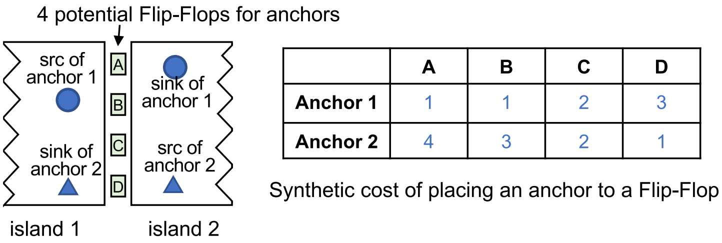
Our LP placement scheme for the anchors is on average 20× faster than the commercial placer and the timing quality is similar.

5 CLOCK ROUTING

5.1 Problem Description

After we finalize the placement of the islands and anchors, we next aim to route the islands in parallel. Since all inter-island connections are anchored, we only need to route each island to

[1] Typically, an anchor region requires 1 to 3 FF columns, about 1/25 the width of an island.



	A	B	C	D
Anchor 1	1	1	2	3
Anchor 2	4	3	2	1

Synthetic cost of placing an anchor to a Flip-Flop

Fig. 9. Illustration of the anchor placement formulation.

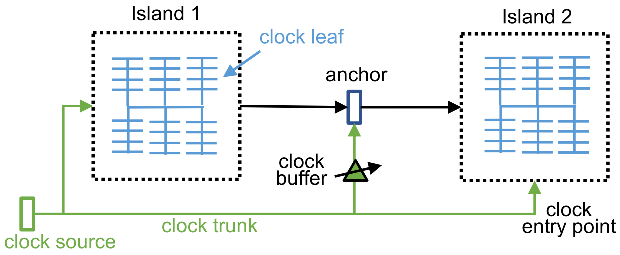


Fig. 10. Route different segments of the clock separately and maintain a stable clock skew in one pass. Step 1: Route the clock trunk. Step 2: Lock the delay level of the clock buffers for anchors. Step 3: Route each island and merge with the clock trunk.

connect to its surrounding anchors. However, we need to take special care of the clock signal because it is a global net that fans out to all islands.

5.2 Challenges and Previous Approaches

Clock routing and data signal routing are interdependent. In a general non-split routing process, the router will first generate an initial clock tree and then route all the data signals. Later, the router may adjust the clock tree for timing optimization.

However, when we route standalone islands separately, the router is unaware of the final clock tree for the entire design. If the island is routed under a different clock tree compared with the final clock tree, the variation in the clock skew will cause timing degradation as well as hold violations. Consider a simple example where the clock signal may enter an island either from the left side or the right side. If the island is routed assuming the clock is from the left, but the actual clock signal arrives from the right in the final stitched design, then the variation in clock skew will cause timing degradation.

A common solution is to first route each island using estimated clock delays and skews; after all islands are combined, the router will globally finalize the clock and re-route the islands to deal with clock skew variations [75]. However, this approach requires an additional global routing step that compromises the compile time.

To address this challenge, we propose dedicated clock management steps to ensure a consistent clock skew before and after the stitching process. Our clock routing flow consists of three steps, which are elaborated in the following subsections. Figure 10 visualizes the key concepts in our clock management scheme.

5.3 Routing the Clock Trunk (S_{10})

The goal of this step is to route from the clock source to the clock entry points of each island. We refer to this route segment as the *clock trunk*. Here, we aim to minimize the clock skew among

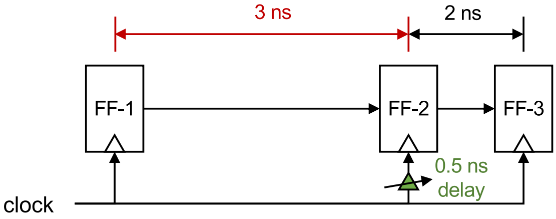


Fig. 11. By introducing an artificial clock delay of 0.5 ns to FF-2, the critical path is reduced from 3 ns to 2.5 ns.

those entry points. To do so, we first route the clock signal from the clock source to the geometry center of all islands. From there, we fan out the clock to reach all islands while minimizing the skew. The obtained clock trunk will be used to constrain the clock routing of each island.

5.4 Locking the Clock Buffers for Anchors (S_{10})

With the clock trunk, we have determined the clock entry points for each island. Since two adjacent islands will route to the same set of anchors in between, we need to *disable* the time-borrowing optimization [23, 25, 79] on the anchor registers to prevent clock skew variations of inter-island paths.

In modern FPGAs, the clock network is equipped with buffers that have configurable delay levels to fine-tune the clock skews [39, 76]. The time-borrowing optimization can utilize the configurable buffers to redistribute the timing slack between consecutive pipeline stages, as demonstrated by Figure 11.

In our flow, we separately route two adjacent islands that connect to the anchors between them. The two independent router processes may result in different time-borrowing schemes and, thus, different clock buffer configurations for the shared anchors. Such potential inconsistency in the clock delay levels for the shared anchors will cause unpredictable timing degradation when the islands are stitched together in the final phase.

To prevent this potential issue, we lock the delay level to the default value for all clock buffers associated with anchor registers.[2] To mitigate the negative impact of this disabled optimization, two aforementioned techniques are beneficial: (1) the source and sink of each anchor net are both pipelined; and (2) the local placement optimization is performed after fixing the anchor locations (S_8).

5.5 Routing and Merging the Local Clocks (S_{11})

With the setup from the previous steps, we are ready to route each island (S_{11}). We enforce the constraint that the local clock net starts from the predetermined entry point and prevent the clock buffers for anchors from being adjusted. A routed island will contain a complete clock route, including the clock trunk. During the final island stitching, redundant clock trunks are unified (S_{13}).

Summary. The clock management steps (S_{10}, S_{11}) ensure that the clock skew remains consistent before/after we stitch the islands together. Since the clock entry points within an island are the same before and after the stitching, the clock skew for intra-island timing paths will remain unchanged. In addition, since we lock the delay level for the anchor registers, the clock skew for inter-island timing paths is also stable. Section 10 shows that without the clock management, we will run into severe hold violations; meanwhile, the measured impact of this method on the achievable frequency is negligible.

[2] In Vivado, this can be achieved by setting the `FIXED_ROUTE` property of the clock net.

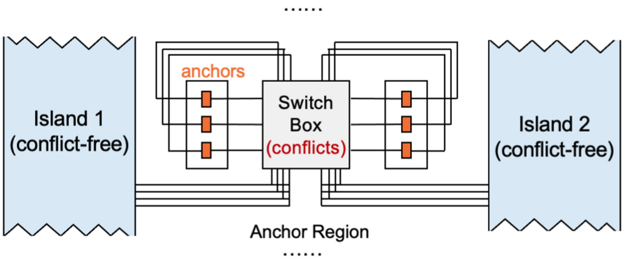


Fig. 12. Detailed view of anchor region. Only one switch box is shown.

6 STITCHING AND INTER-ISLAND ROUTING

6.1 Island Merging (S_{12}, S_{13})

In the previous sections, we present how to place and route the islands in parallel. As a result, we will obtain separate post-routing checkpoints, each for one island. Next, we need to assemble them together into the complete physical implementation. While this step is conceptually simple, it is not supported by the off-the-shelf commercial tools. We utilize the open-source RapidWright framework [42] to edit the netlists and assemble the physical information of the island checkpoints.

The checkpoint of each island also includes its surrounding anchor registers. Thus, when we stitch the netlists together, we need to unify (or merge) the duplicated anchor registers, as the same anchor is included in the checkpoints of both islands on its two sides. Since the physical information of the duplicated anchors is consistent after the parallel placement (Section 4), we can safely merge them without causing conflicts in anchor locations. Further, our clock routing scheme (Section 5) ensures that different islands are routed under the same clock trunk; thus, the clock net can also be merged without conflicts (S_{13}).

6.2 Inter-island Routing (S_{14})

After the individual checkpoints are assembled together, we need to resolve the routing conflicts in the anchor regions. This is the last step of the RapidStream flow.

Problem Description: Figure 12 shows the low-level routing resources in the anchor region and why routing conflicts may arise. Since the switch boxes in the anchor region are shared, the two router processes may both exploit the same physical wire segments when they separately route islands 1 and 2. According to our profiling, the conflicting nets in the anchor region amount to 5% to 10% of all the nets. Those conflicts will be exposed after we glue the post-routing checkpoints of islands together.

One potential solution is to resolve the inter-island conflicts pair by pair. Figure 13 illustrates why this will not work. In Figure 13, we could try to separately reroute the conflict nets between islands (1, 2) and between islands (2, 3). However, while pairwise rerouting resolves the anchor region conflicts, it will lead to new conflicts within the islands. In Figure 13, assume the black and the yellow nets are separately routed by two router processes; conflicts may show up inside the islands (the red segment).

Therefore, we have to do a global routing pass to fix the inter-island conflicts. We present two solutions for this routing task. One set of experiments uses the Vivado router in order to maintain the best performance. The other solution relies on a customized open-source routing solution for the best compile time.

Commercial routers can resolve the inter-island conflicts at the expense of some runtime overhead because they are optimized for general-purpose routing. The Vivado router spends about 1/4

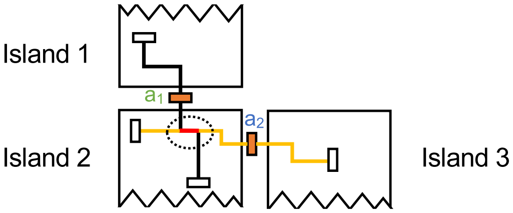


Fig. 13. Pairwise inter-island routing will not work because it may cause conflicts inside the island.

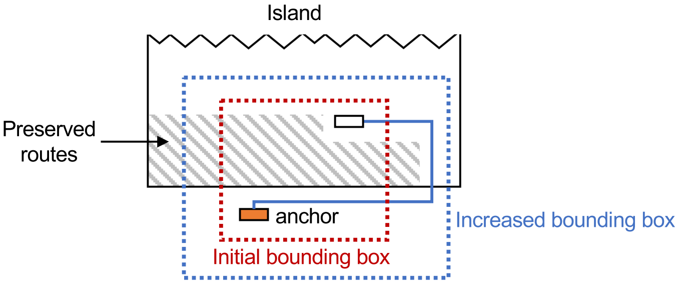


Fig. 14. Required long routing detours outside of the initial net bounding box.

of the time for initialization, 1/4 of the time for the actual routing and timing closure, and 1/2 of the time looping through a set of optimization steps even after timing closure.

However, our routing problem has two unique features. First, 90% to 95% of the nets (intra-island) are fully routed and have been well optimized for timing. Second, the conflicts are clustered in the anchor region between islands. In this case, we can potentially utilize the special properties of the problem for further speedup.

7 ACCELERATE ROUTING WITH CUSTOMIZED PARTIAL ROUTER (RAPIDSTREAM 1.0)

For this unique problem, we build a lightweight *partial router* that only rips up and reroutes the conflicting nets from/to the anchor regions. The partial router preserves other fully routed nets, that is, masking the routing resources used by those nets and skipping any processing on those nets.

One challenge of preserving the non-conflicting nets is how to determine suitable sizes for the *bounding boxes*. During the routing process, the bounding boxes restrict the accessible routing resources for the net. Usually, their sizes are determined based on the pin locations of a net. A large bounding box allows more flexibility for the net but will incur extra runtime, whereas a small bounding box limits the routability but also reduces the route time. In a typical routing process with no preserved nets, the effective bounding boxes for all nets could be determined in advance and will remain fixed during routing [29, 33, 48, 66, 90]. However, the conventional approach does not work in our situation due to the reduced routing flexibility after we preserve all the intra-island nets.

Figure 14 shows a case in which a net needs long horizontal routing detours outside of its bounding box. This is because there is resource blockage within the initial bounding box resulting from the preserved nets. Without expanding the bounding box, the net cannot be routed. There are also cases in which vertical long routing detours are needed for successful routing. Therefore, it is difficult to determine suitable bounding boxes for all the target nets before routing.

To address this issue, we use a simple heuristic to start with small bounding boxes and incrementally increase the box size. Starting from the second iteration, our router expands the four sides of the bounding box for each net that will be ripped up and rerouted.

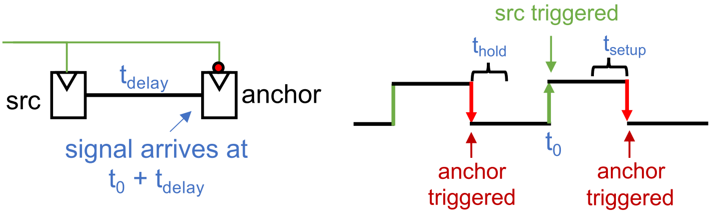


Fig. 15. Anchors triggered on negative clock edges.

We achieve the goal by customizing an open-source router called RWRoute [89]. We upgrade its partial routing function to be timing driven and enable the tool to expand bounding boxes at runtime. With a single thread, our customized router achieves a 4× speedup compared with the Vivado router.

As of now, RWRoute relies on an open-timing model [51] to achieve timing-driven routing. However, this model provides only the *slow path* delay estimation of routing resources. As a result, RWRoute could not resolve hold violations, which require the *fast path* delay estimation of the routing resources. We present a temporary workaround in the next section to eliminate hold time requirements at the expense of some performance.

Workaround for Hold Violation in Solution 2: Since the customized RWRoute will only route the nets to/from the anchor registers, we make all anchor registers to be triggered by the negative clock edge, for example, in Figure 7, and modify the registers in the green box to be triggered by the *negative* clock edge while keeping everything else triggered by the *positive* clock edge.

Figure 15 depicts the idea of when the anchor is the signal sink. The same reasoning applies when the anchor is the signal source. Assuming a zero clock skew, the source FF is triggered at t_0 and the anchor FF is triggered at $t_0 + t_{period}/2$ to transfer Signal i. The signal will arrive at the anchor at $t_0 + t_{delay}$. For Signal i to be properly captured at the anchor FF while still not interfering with the capturing of Signal $i - 1$, both Equations (2) and (3) must be satisfied:

$$t_0 + t_{slow\_delay} < t_0 + t_{period}/2 - t_{setup} \tag{2}$$

$$t_0 + t_{fast\_delay} > t_0 - t_{period}/2 + t_{hold}. \tag{3}$$

Equations (2) and (3) can be reduced to (4) and (5):

$$t_{period} > 2(t_{setup} + t_{slow\_delay}) \tag{4}$$

$$t_{period} > 2(t_{hold} - t_{fast\_delay}). \tag{5}$$

Therefore, with negatively-triggered anchors, we can always increase the clock period to satisfy the conditions and, thus, avoid any setup/hold violation on the anchor nets when RWRoute reroutes them to fix conflicts in the anchor region. Meanwhile, the intra-island nets are routed by Vivado and are free of hold violation.

Note that this technique of clock phase shifting is a temporary measure, which will no longer be needed if an open fast-path timing model is provided. This experiment shows us the potential for the best runtime and advantages of an open-source partial router.

8 PRE-PARTIAL-ROUTING OF INTER-ISLAND NETS (RAPIDSTREAM 2.0)

The divide-and-conquer approach used in Rapidstream 1.0 requires a combining phase, which is time-consuming. Since the RWRoute approach in the previous section does not yet support hold-time modeling, we have to rely on a standard router for the combining phase, which becomes

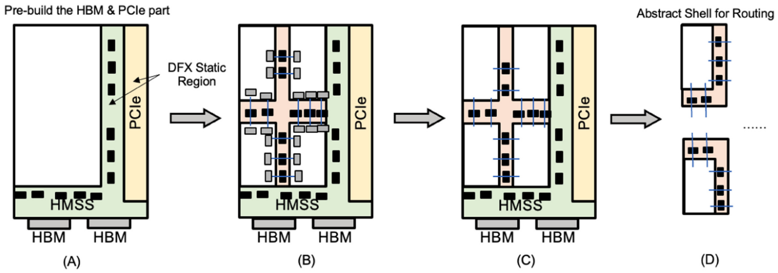


Fig. 16. Partial routing of the inter-island nets using a skeleton design. We first do a complete routing of the nets from the anchor registers to the source/sink cells inside the island, then we use RapidWright to prune away most routing nodes inside the island and leave the net in an antenna state. The endpoints of the inter-island nets are viewed as virtual partition pins. Later when we route the island, the router will connect the island cells to those partition pins.

a compile-time bottleneck. In this section, we present a different approach to accelerate routing that generates fully legal routing results. Instead of fixing routing conflicts in the anchor region after routing the island, we partially route the anchor nets with the assistance of the RapidWright framework [42]. Figure 16 shows the process.

Figure 16(A) shows the pre-built shell that includes the PCIe, DMA, and HBM subsystem IPs. Then, in Figure 16(B), we partition the unused (white) area into the anchor regions (light red) and disjoint island regions (white). To route the inter-island nets efficiently, we prune away the majority of the logic elements in the island region and only preserve the source and sink nodes for the anchor registers. This significantly speeds up the routing of the inter-partition nets.

Instead of preserving the full route of the inter-partition nets, we trim away the majority of the routing of each inter-island net and only preserve the part between the anchor register and a virtual partition pin inside the island, as shown in Figure 16(C). In this way, we can later route each island independently and connect the inner-island logic to the virtual partition pins without causing a conflict in the anchor region. The routing of each individual island could be further accelerated using the abstract shells shown in Figure 16(D), which prunes away all irrelevant elements and only preserves the cells that directly connect to an island.

8.1 Avoid Routing Conflicts

Now that we are partially routing the inter-island nets using a skeleton design, the router will not consider the routability of other placed elements inside the island. Therefore, it is possible that the routing results with the skeleton design will not be compatible with the full island because the routes of the anchor nets may block the only way to reach certain resources. Specifically for AMD/Xilinx FPGAs, certain routing nodes have several outputs, which we refer to as *multi-fanout nodes*. If we use such a routing node between the anchor register and the virtual partition pin, the logic elements at the other outputs of the node will be blocked and become unroutable. We present how we use the RapidWright framework [42] to eliminate such situations.

Figure 17 shows an example. To route from the source FF to the sink FF, three routing nodes are used. However, node 2 is a multi-output node that has one input and two outputs. If node 2 is used for the net, then the net connecting to the blocked FF (shown in red) cannot be routed without causing conflicts. Worse, if node 2 is the first node that spans into the island region, then it will always be preserved in the partition pin selection process. In a normal flow in which the router has full knowledge of the design placement, the router will avoid using such nodes. But in our

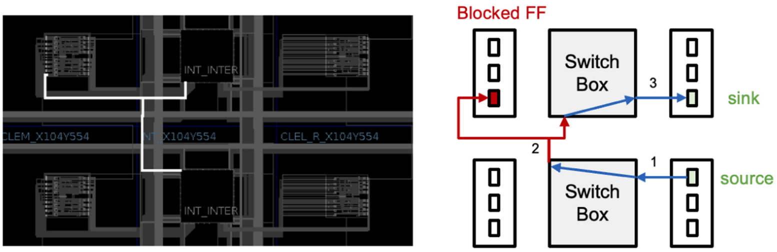


Fig. 17. Example of a route with a multi-output node. The red FF is made unreachable by other nets since routing node 2 has been occupied.

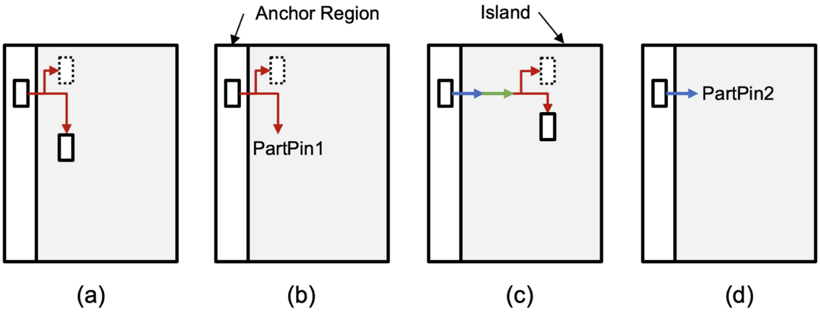


Fig. 18. Enforcing a gap at the island boundaries to prevent routing conflicts. In (b), the special multi-output node will directly connect to end cells, thus, blocking certain locations. In (c), since the sink cell is forced to be away from the island boundary, multiple routing nodes (blue, green, red) are used to reach the sink cell. Multi-output nodes (red) will only be used for the final connection to the end cell. Other single-output routing nodes (blue and green) will not block other cells. In (d), we prune the routes and only preserve an entry point to the island as the partition pin.

situation, we prune away the majority of the island logic to speed up the routing of inter-island nets; thus, the router has no knowledge of whether a multi-output node will cause conflicts.

To address the issue, our solution is to reassign the partition pin with RapidWright and at the same time leave a gap of one column or row between the anchor region and the island region. Figure 18 explains our approach. For routing structures of the AMD/Xilinx FPGAs, such multi-output nodes are used for nets to reach the end cells. In other words, such nodes will only appear towards the sink of a net (Figure 18(a)). If the sink cell is very close to the island boundary, then very likely such a multi-output node will be the first node that reaches the island region from the anchor region. Therefore, we have no other choice but to select this node as the partition pin (Figure 18(b)). As a result, the cells at the other outputs of the node will be blocked. However, if the end cell is constrained to be deeper inside the island, then we can guarantee that the first node reaching the island region will be a single-output node (the blue node in Figure 18(c)). Thanks to the RapidWright framework, we are able to locate the first routing node inside the island region and select it as the partition pin, which is not possible in the standard Vivado (Figure 18(d)). Since a single-output node will not directly connect to a cell, we ensure that the partition pin blocks no cells.

Another situation is shown in Figure 19. The Super Long Logic (SLL) nodes for die boundary crossing are shared among multiple connections. If a net occupies the SLL node in its routing, then certain Laguna FFs will be unreachable. Therefore, we adopt a similar approach to enforce a gap around the Laguna columns.

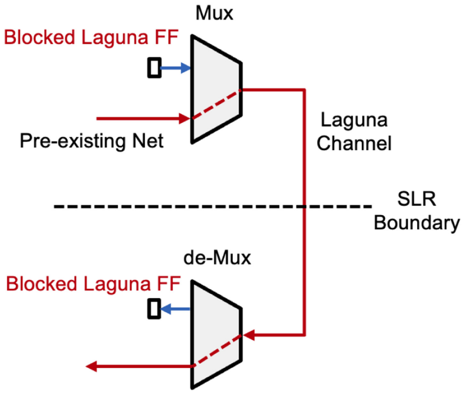


Fig. 19. Example of a route with an SLL node. The red FFs are made unreachable since the SLL node is the only input/output connection to them.

Problems like this will not occur in a conventional routing process in which the router has full knowledge of all placed elements and the router will prevent a route to block other elements as the first priority. However, in our flow, the routing of inter-island nets is performed out of the placement context; thus, we must add extra constraints to prevent such problems. In the actual routing process, multi-output nodes are mostly used to connect the endpoints to the switch boxes. Thus, we enforce a gap of one row of resources between the island region and the anchor region during placement. Meanwhile, we will pre-collect all the SLL nodes occupied by the static shell using RapidWright [42] and mark the corresponding Laguna registers as prohibited in placement.

9 COMPARISON OF RAPIDSTREAM 1.0 AND 2.0

The core difference between the two versions is the order between routing intra-island nets and routing inter-island nets. From an algorithmic perspective, there is not much of a difference and both methods should work. The major reason that pushes us for the change is whether the split-compilation flow could be efficiently supported by existing tools. As we do not have a router that could quickly fix the local routing conflicts (RWRoute can, but it does not support hold-time modeling yet), we explore and pivot to the 2.0 fashion that utilizes a standard router in a much more efficient way to achieve our goal.

Another practical advantage of RapidStream 2.0 is that we can enable more reuse of robust components of a standard FPGA CAD flow. As we switch to routing the inter-island nets first, RapidStream 2.0 could be built on top of the AMD/Xilinx Partial Reconfiguration flow, which is now branded as Dynamic Function eXchange (DFX). The anchor region will become the static region and each island will become a dynamic region in the DFX flow. The PCIe and HMSS parts are also part of the static region, and we utilize the nested DFX feature to create multiple layers of static regions. The DFX flow will ensure the isolation of the static and the dynamic region and performs clock management as an inherent step, which makes our flow more robust. Moreover, Rapidstream 2.0 uses the *abstract shell* feature of DFX to quickly setup a customized routing environment for each island and prunes away irrelevant logic elements that are not directly connected to the island. As a result, routing an island in an abstract shell is significantly faster than routing in a full shell. It is through the DFX environment that we are able to integrate RapidStream 2.0 with the AMD/Xilinx Vitis accelerator workflow so that we can reuse the host–device communication infrastructure to execute our generated bitstream on a real FPGA board.

10 EVALUATION OF RAPIDSTREAM 1.0

10.1 Implementation Details

We implement the key modules of RapidStream 1.0 in Python with approximately 8K lines of code (LoC). We evaluate RapidStream using four servers, each with the 56-core Intel Xeon E5-2680 v4 CPU at 2.40 GHz and 128 GB of memory. All servers use the Ubuntu 18.04 operating system. In our experiments, we target the Xilinx UltraScale+ U250 FPGA, which consists of four dies that are stacked vertically. The target frequency is 400 MHz (i.e., a clock period of 2.5 ns). The CAD tools used in the RapidStream flow are summarized as follows.

Phase 1: We use Vivado HLS 2020.1 to generate the initial RTL, then RapidStream floorplans the HLS dataflow design (S_1, S_2). Based on the floorplanning results, RapidStream post-processes the RTL generated by Vivado HLS to insert the inter-island pipelines (anchor registers) and rebuilds the RTL hierarchy for each island (S_3–S_5).

Phase 2: We use Vivado 2021.1 to synthesize each island (S_6). During placement, we first use Vivado (`place_design`) to get the initial island placement (iteration 1, S_7). Then, we use our ILP-based method to place the anchors (iteration 2, S_8). Finally, we switch back to Vivado (`phys_opt_design`) to incrementally optimize the placement of islands (iteration 3, S_9). In island routing (S_{10}, S_{11}), we pre-build the clock trunk and lock the clock buffer (set_property `FIXED_ROUTE`)[3] for anchors (S_{10}), which are passed as constraints to the Vivado router (S_{11}). We use the "Explore" directive in Vivado so that the tool will spend more time in the optimization process.

Phase 3: We build a stitcher based on RapidWright to edit the netlist of islands and put them together (S_{12}, S_{13}). We then use Vivado for inter-island routing (S_{14}). We separately compare Vivado and our timing-driven partial router RWRoute on S_{14}.

Island Organization: We currently employ an empirical scheme to organize the U250 FPGA fabric as 32 islands in eight rows (four islands per row), where each island has a uniform height of 120 CLBs.[4] Between adjacent islands, we reserve 3 empty columns (or 10 rows for vertically adjacent islands) of CLBs as the anchor region to accommodate the anchor registers. The width of the anchor region is approximately 1/25 that of an island. At die boundaries, we use all Laguna columns as the anchor region (see Figure 5).

Two-Level Stitching: Specifically for Xilinx UltraScale+ devices, we employ a two-level method in Phase 3. We first stitch the island-level checkpoints into die-level checkpoints and route the inter-island nets; we then stitch together all the die-level checkpoints into the final checkpoint. Note that in the second stitching step, the die-level checkpoints can be readily assembled without any rerouting. As shown in Figure 5, the anchor regions at the die boundary of the Xilinx UltraScale+ FPGAs are different, where the islands on the two sides of the die boundary rely on the dedicated Laguna channels for cross-die signals. Since the actual wires within the channel are point-to-point and separated from each other [74], there are no conflicts when die-level checkpoints are merged.

Distributed Execution: Each step of RapidStream is launched as soon as its input is ready. For example, the placer process for an island will start immediately after the corresponding synthesis process has finished, and no synchronization is needed to wait for all synthesis processes to complete. Likewise, the process to optimize the island placement will start as soon as the dependent anchor placement processes have exited and all surrounding anchors have been placed.

[3] Please refer to our code for more details.

[4] Each CLB in Xilinx FPGAs contains 16 FFs. Note that the width of islands may vary slightly based on clock region boundaries.

Table 1. Benchmarks

Name	# V	# E	Topology	DSP %	BRAM %	FF %	LUT %
MM	463	854	2-D Mesh	62	23	34	69
CNN	439	813	2-D Mesh	59	33	32	50
LU	1691	4483	Triangular	20	41	26	66
MTTKRP	360	760	2-D Mesh	66	33	30	48
2-D Stencil	266	1562	Irregular DAG	52	21	27	45
3-D Stencil	1314	2866	Irregular DAG	64	39	35	53

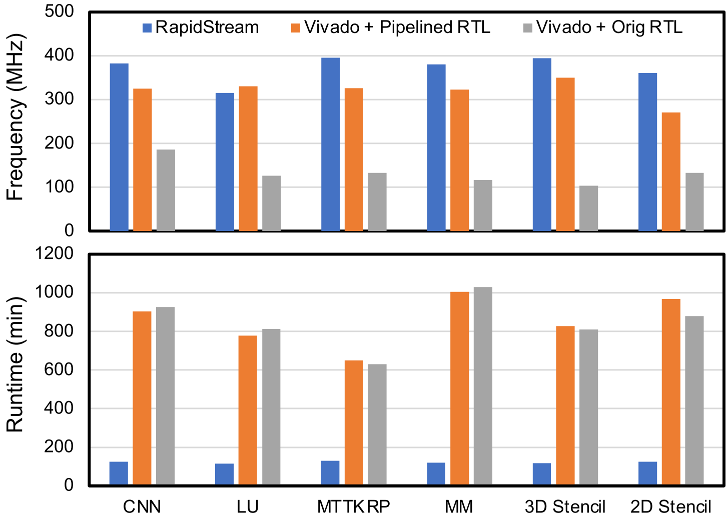


Fig. 20. Comparison of the runtime and achievable frequency between RapidStream and Vivado.

10.2 Benchmarks

To evaluate RapidStream, we use six large-scale dataflow designs, listed in Table 1. We denote the number of PEs as “#V” and the number of FIFO connections between PEs as “#E”. The matrix multiplication (MM), CNN, L/U decomposition (LU), and MTTKRP are from the AutoSA project [67]; the 2-D and 3-D stencil accelerators are from the SODA project [11].

The benchmarks are mapped onto the target U250 FPGA, which contains 5,376 BRAMs, 12,288 DSPs, 3,456K FFs, and 1,728K LUTs. The mapped designs consume 60% to 70% of the available resources.

10.3 Runtime Reduction

Figure 20 shows the comparison of runtime and the achievable frequency between the vanilla Vivado flow and RapidStream. Since RapidStream will insert additional pipelining to the RTL, we consider two Vivado baselines: (1) the original RTL generated by HLS and (2) the version that has been pipelined by RapidStream.

By default, we use Vivado for inter-island routing (S_{13}) to pursue the best timing quality. In this case, we achieve a 5 to 7× speedup and reduce the otherwise >10-hour compile time to around 2 hours.

In terms of frequency, we achieve better results than both baselines. Since each island is much smaller than the entire design, Vivado can better optimize the timing of each island. The only

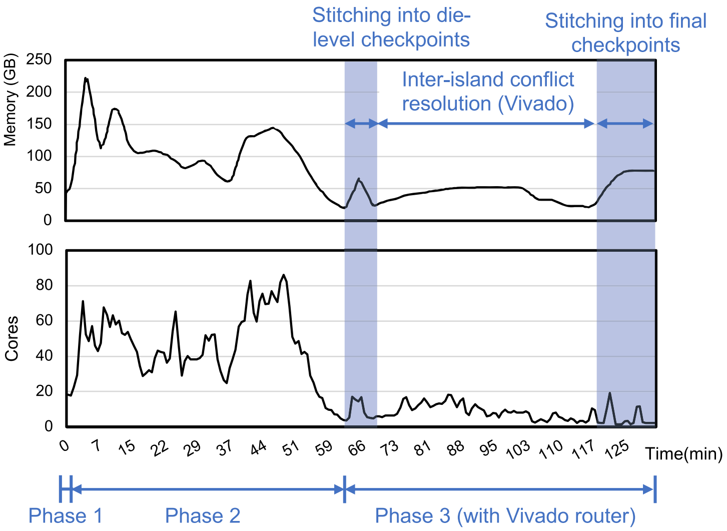


Fig. 21. CPU and memory usage of the RapidStream run on the CNN design. No reroute needed after die-level stitching (Section 10.1).

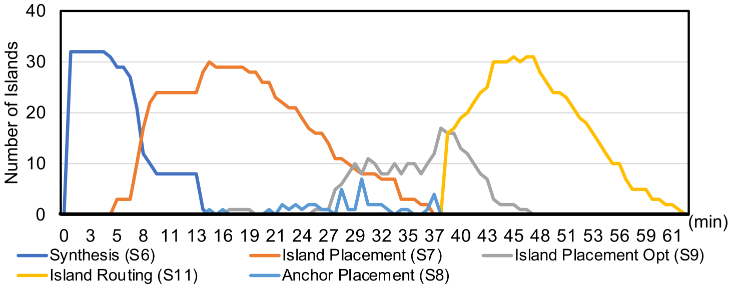


Fig. 22. Number of active jobs in Phase 2.

exception is the LU benchmark, which has many division operations that become the critical paths in both flows.

Figure 21 shows the CPU and memory utilization when we use RapidStream to compile the same CNN design as in Figure 1. While Vivado uses 2.1 cores on average and runs for about 14 hours, RapidStream uses 26 cores on average and runs for about 2 hours.

Figure 22 breaks down the parallel compilation process of Phase 2 for the CNN design by plotting how many islands are active in each step at a given time. For example, after 11 minutes, there are 24 islands in synthesis while 8 islands have started placement. Notably, the asynchronous execution of RapidStream alleviates the load imbalance issue within each step.

10.4 Fast Inter-island Routing

Figure 21 shows a long tail in compile time during Phase 3, where we use Vivado to resolve the inter-island routing conflicts. As mentioned in Section 6.2, we customize the open-source RWRoute to further accelerate this step. Figure 23 shows the comparison between using the customized RWRoute and using Vivado for S_{14}.

On average, we achieve a 4× speedup over the Vivado router, reducing the conflict resolution time from about 25 minutes to 6 minutes. The RWRoute flow achieves a lower frequency as it

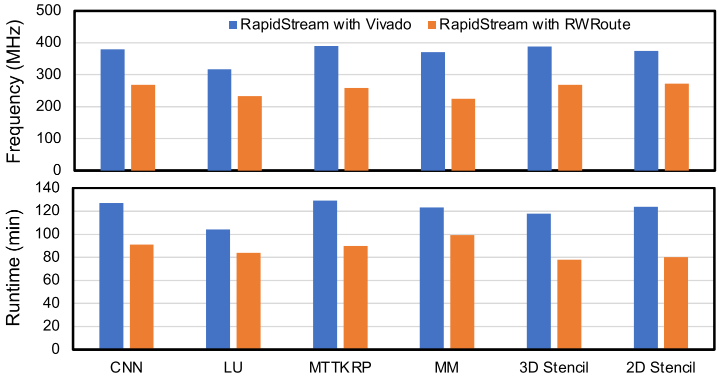


Fig. 23. Runtime comparison in conflict resolution.

relies on negatively triggered anchors (see Section 7) to prevent hold violations, which sacrifices the setup slack. This performance loss can be avoided if a timing model with fast-path delays is available.

In addition to reducing routing time, we further minimize the unnecessary interactions between Vivado and RapidWright through reading/writing checkpoints. Since our custom router is also implemented under the RapidWright framework, we can directly pass the stitcher's output in memory to RWRoute. This can also alleviate the long tail issue in the compile time of Phase 3. By our projection, we can reduce the end-to-end time reported in Section 10.3 down to ∼80 minutes, which is a 7 to 10× speedup over the Vivado flow.

10.5 Anchor Placement

In our three-iteration approach to placing the islands and anchors (S_7–S_9), we propose a min-cost matching formulation for the anchor placement (iteration 2, S_8). We use the MM benchmark to compare our lightweight placer with the Vivado placer. With 32 islands, there are 52 island pairs and we will have 52 placer processes, each of which handles one pair of islands.

In terms of speed, the min-cost matching placer takes less than a minute to place the anchors between pairs of islands, whereas it takes Vivado 21 minutes on average (including the time to read the checkpoints). As for the timing quality, both placement schemes can achieve above the 2.5-ns target period after three iterations, as shown in Figure 24. Note that the timing report is based on placement-level timing estimation by Vivado.

In some cases, our min-cost matching placement even achieves higher setup slacks than Vivado. This is because our min-cost matching formulation will always place the anchors at die boundaries onto the die-crossing channels to balance the signal delays on two sides. However, Vivado often places the anchors outside the die-crossing channels as the timing target is still met.

After we place all the anchors (iteration 2), we will perform local optimization of the island placement (iteration 3). We measure the setup slack of all nets from/to anchors to check the placement quality of our min-cost matching placement formulation. Based on Vivado's timing report, the average setup slack of anchor nets after iteration 2 is 0.55 ns (when targeting 2.5 ns or 400 MHz), while iteration 3 improves the average slack to 0.69 ns.

10.6 Clock Management

Here, we demonstrate the advantages of preserving the clocking trunk using a number of experiments with the MM benchmark. Figure 25 shows the timing degradation when we stitch the islands

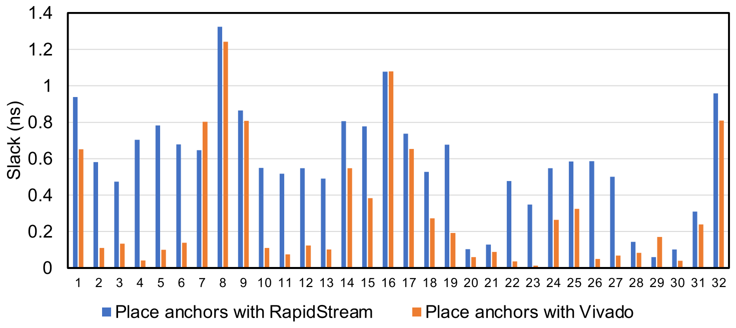


Fig. 24. Post-placement slack between using the Vivado placer or the min-cost matching placer for anchor placement.

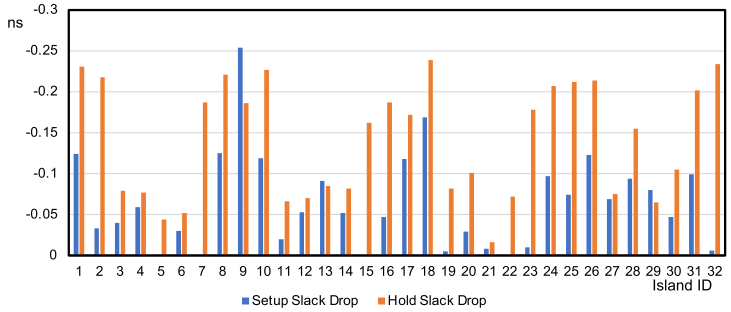


Fig. 25. Timing loss after stitching without clock management.

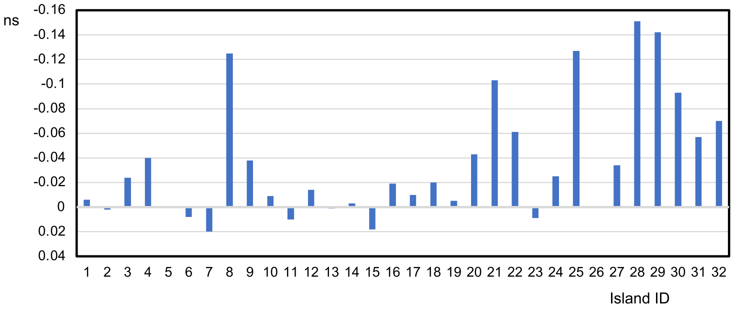


Fig. 26. Clock preservation reduces timing degradation.

together and route the clock net afterward. In this case, we route each island without preserving the clock trunk. The router relies on an estimation of the clock skew when routing the data signals. As a result, the actual clock skew after stitching may be different. As shown by the figure, all islands run into hold violations after stitching. Notably, the setup/hold slack times deteriorate by about 0.25 ns for the islands, which will almost always cause hold violations.

Figure 26 shows the setup slack differences when an island is routed with the preserved clock trunk. This is compared to the reference case used in Figure 25 without any clocking constraints. The drop in setup slack is at most 0.15 ns, which is much smaller than that in Figure 25. The key takeaway is that we avoid the setup/hold loss during stitching by keeping the clock consistent.

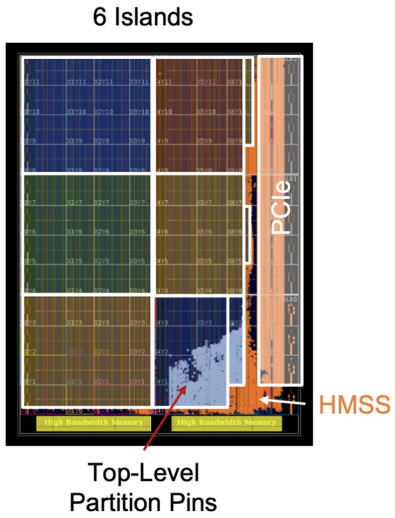


Fig. 27. An example shell for RapidStream 2.0 corresponding to Figure 16(C).

11 EVALUATION OF RAPIDSTREAM 2.0

In this section, we describe the implementation and evaluation of RapidStream 2.0.

11.1 Implementation Details

RapidStream 2.0 takes a TAPA [30] dataflow program as input. Compared with Version 1.0, the benefit of using a TAPA program as input is that the TAPA compiler will help check whether the input is a valid dataflow program. It can also take in a pre-built shell that exposes AXI interfaces to the TAPA dataflow program. In our prototype, we build a shell that uses 4 HBM channels on the AMD/Xilinx U280 FPGA, shown in Figure 27. The rightmost block is provided by the Vitis framework, which includes the PCIe and DMA module. The logic surrounding the PCIe block is the HBM subsystem that instantiates 4 HBM channels. For now, we adopt a fixed device partition strategy in which the FPGA is divided into 6 islands. We will discuss our plan in the future work section to make the shell more general. All computation jobs are executed on only one Intel Xeon server with 16 physical cores and 256 GB of memory.

11.2 Benchmarks

We have evaluated RapidStream 2.0 with two benchmark designs that are compatible with our prototype shell with 4 HBM channels enabled. The two designs implement accelerators for stencil computation and are generated using the SODA compiler. Due to the different requirements on DSPs, SODA adopts different topologies for the two designs. Table 2 shows the details of the two designs. Table 3 shows a detailed comparison of the compile time between RapidStream 2.0 and Vivado. For each benchmark design, we run three groups of experiments: (1) compile by the vanilla Vivado, (2) compile by Vivado but with the floorplan guidance from AutoBridge [30], and (3) compile by RapidStream 2.0. The table records the time of each major step and the final frequency.

Compared with the vanilla Vivado flow, RapidStream 2.0 is 7.5× and 5.1× faster on the two designs while achieving even higher frequency. RapidStream 2.0 could achieve more than 300 MHz, but in our prototype shell, we only set the clock to 300 MHz. We observe that adding floorplan hints to Vivado could slightly reduce its optimization time, but RapidStream still achieves 5.7× and 4.3× speedup with the same final frequency.

Table 2. RapidStream 2.0 Benchmarks

	LUT	FF	DSP	BRAM	# Task	# FIFO
gaussian-int	32%	14%	27%	46%	840	2440
gaussian-float	52%	40%	77%	13%	232	648

Table 3. Detailed Comparison Between RapidStream 2.0 and Vivado

Test\Time (min)	Synthesis	Placement	Routing	Total Time	MHz
gaussian-int-vivado	74 (6.2×)	131 (6.6×)	243 (4.6×)	455 (5.1×)	268
gaussian-int-vivado-autobridge	80 (6.7×)	93 (4.7×)	143 (2.7×)	386 (4.3×)	300
gaussian-int-rapidstream	12	20	53	89	300
gaussian-float-vivado	196 (9.6×)	236 (6.7×)	454 (7×)	897 (7.5×)	237
gaussian-float-vivado-autobridge	208 (10.4×)	220 (6.3×)	248 (3.8×)	683 (5.7×)	300
gaussian-float-rapidstream	20	35	65	120	300

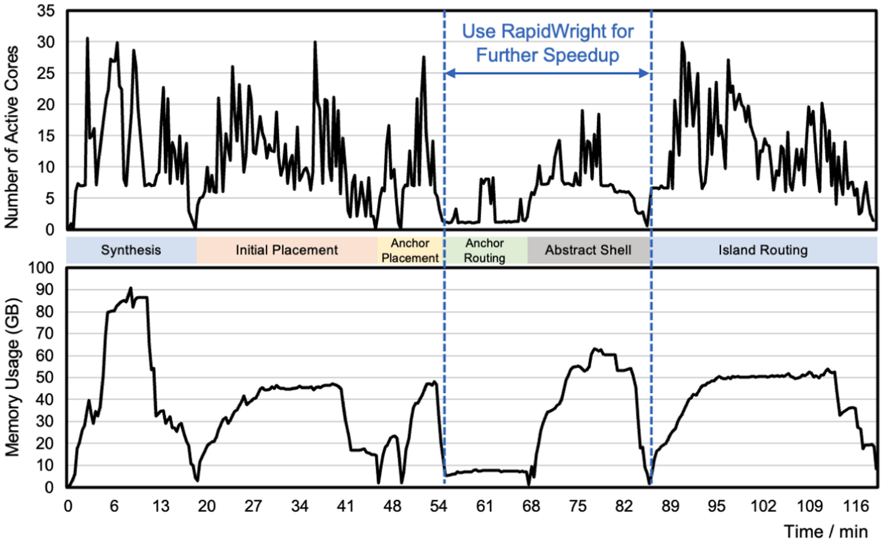


Fig. 28. Profiling of the CPU and memory usage in RapidStream 2.0 for the gaussian-float benchmark.

Note that the speedup on routing is noticeably smaller than on placement. This is because we have the extra overhead to route the inter-island nets and create the abstract shells. A more detailed time breakdown is shown in the next subsection.

11.3 Profiling of RapidStream 2.0 Compilation

Figure 28 shows the CPU and memory usage of the RapidStream 2.0 compilation process. On average, 10.3 CPU cores are active and the peak memory usage is around 90 GB. The metrics are different compared with RapidStream 1.0 because the 2.0 prototype only partitions the design into 6 islands.

As we construct a skeleton design for the routing of inter-island nets, the time of this step is acceptable even with a standard router (about 15 min). We are in the process of updating RWRoute for this task and our initial profiling shows that we can reduce this step to around 3 minutes. Currently, the abstract shell generation also takes around 15 minutes. It remains future work to speed up this step with a customized logic pruning tool based on RapidWright.

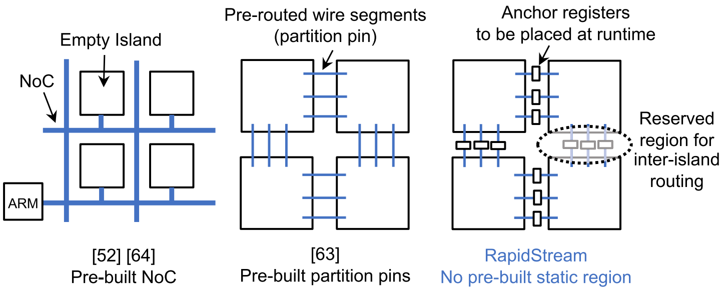


Fig. 29. Comparison between previous works and RapidStream.

Still, the routing overhead in RapidStream 2.0 is significantly lower than in RapidStream 1.0. We test the gaussian-float design using the RapidStream 1.0 flow. RapidStream 1.0 needs almost 3 hours to fix the inter-island routing conflicts after routing the islands, whereas RapidStream 2.0 routes the inter-island nets first with a skeleton design, which only requires about 15 minutes for routing and 15 minutes for abstract shell generation. As a result, RapidStream 2.0 is almost 2× as fast as RapidStream 1.0 for this design. Again, we want to note that such a difference is only due to practical engineering factors and RapidStream 1.0 could be just as efficient if the router is optimized to local routing fixes.

12 RELATED WORK

Split Compilation for HLS Designs can exploit the flexibility to introduce additional pipelining when appropriate, which is in contrast to the split compilation methods for *RTL* designs (see Section 1).

Previous efforts on HLS-level split compilation are based on pre-building a fixed *static region* to divide the FPGA into islands. The static region includes pre-placed and pre-routed logic that remains unchanged. Then, a design is divided and mapped onto those disjoint islands. The authors of [54, 55, 72, 73] pre-build an NoC based on partial reconfiguration [77], as shown in Figure 29. However, these approaches suffer from the area overhead and the limited NoC bandwidth. Several recent efforts on FPGA virtualization [81–83] also rely on pre-building a static region to form disjoint islands.

In *DW* [69], a static region only consists of a set of *partition pins*, which are pre-routed wire segments at the boundary of two adjacent islands. This helps reduce the area overhead. However, DW needs users to manually change the design and map inter-island nets to the partition pins. Their following work, HiPR [70, 71], can floorplan all the PR regions automatically with the least amount of human intervention from C to bitstreams. Nevertheless, the number and distribution of partition pins are fixed in both DW and HiPR, making timing closure more difficult. While DW and HiPR are capable of reaching frequencies of up to 187 MHz and 300 MHz, respectively, we can achieve a significantly higher frequency of nearly 400 MHz.

Soft Cores with NoC. One way to reduce the compile time is to implement a collection of soft processors on the FPGA [3, 4, 18, 35, 36, 68, 80] and then connect these processors by a configurable NoC [26, 34, 40, 53, 65]. In comparison, we focus on building high-performance application-specific accelerators.

Acceleration Based on Hard Macros. Researchers have explored acceleration utilizing pre-implemented hard macros [19, 27, 43, 49, 52, 84]. Hard macros consist of pre-built circuitry and can be reused. However, this approach may only cover a very limited portion, if any, of an arbitrary input design. In addition, the fixed shapes of predetermined macros may result in area waste.

Co-optimizing HLS and Physical Design. Guo et al. [30] couples floorplanning with HLS synthesis to pipeline the global data transfer logic. RapidStream (in S_2) also adopts the iterative partitioning floorplan algorithm. AutoBridge and other works that co-optimize the physical design process and the HLS compilation [16, 31, 64, 85–87] rely on the conventional RTL-to-bitstream toolchain.

Dataflow Designs. RapidStream targets the dataflow design pattern, which has been well studied in theory [5, 44] and has been applied in a rich set of application domains, including linear algebra [61, 62, 67], graph processing [8, 12–14], image processing [11, 88], sorting [56–59] and many more. Recently, HLS tools with dynamic scheduling [9, 10, 38] are gaining popularity. They introduce elastic components such as FIFOs to enable a dataflow-style execution, which could potentially be utilized by RapidStream in the future.

13 CONCLUSION

RapidStream is an automated split compilation flow for HLS dataflow designs. It features tight integration of HLS-level pipelining and physical design automation to enable split compilation while maintaining high timing quality. Compared with a commercial toolchain, RapidStream achieves about a 5 to 7× reduction in compile time and up to a 1.3× increase in frequency for HLS dataflow designs. In addition, our results show potential for up to an order of magnitude speedup by leveraging customized open-source routers.

ACKNOWLEDGMENTS

We thank Dina G. Mahmoud for helping with the artifact evaluation. We thank Gurobi and GNU Parallel for their academic licenses. The opinions expressed by the authors do not represent future AMD policies.

REFERENCES

[1] Matthew An, J. Gregory Steffan, and Vaughn Betz. 2014. Speeding up FPGA placement: Parallel algorithms and methods. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.

[2] Melvin A. Breuer. 1977. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*. 284–290.

[3] Davor Capalija and Tarek S. Abdelrahman. 2011. Towards synthesis-free JIT compilation to commodity FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 202–205.

[4] Davor Capalija and Tarek S. Abdelrahman. 2013. A high-performance overlay architecture for pipelined execution of data flow graphs. In *2013 23rd International Conference on Field Programmable Logic and Applications*. IEEE, 1–8.

[5] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (2001), 1059–1076.

[6] Tony Chan, Jason Cong, and Kenton Sze. 2005. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the 2005 International Symposium on Physical Design*. 185–192.

[7] Chandra Chekuri. 2010. (2010). Retrieved from https://courses.engr.illinois.edu/cs598csc/sp2010/Lectures/Lecture11.pdf.

[8] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[9] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 288–298.

[10] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2021. DASS: Combining dynamic and static scheduling in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[11] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, 1–8.

[12] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[13] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending high-level synthesis for task-parallel programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. IEEE, 204–213.

[14] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2022. Democratizing domain-specific computing. *Commun. ACM* 66, 1 (2022), 74–85.

[15] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (Aug 2022), 42 pages. https://doi.org/10.1145/3530775

[16] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality aware transformation for high-level synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, 125–128.

[17] Jason Cong and Yi Zou. 2009. Parallel multi-level analytical global placement on graphics processing units. In *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*. IEEE, 681–688.

[18] James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 13–22.

[19] James Coole and Greg Stitt. 2012. BPR: Fast FPGA placement and routing using macroblocks. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 275–284.

[20] Kaushik De and Prithviraj Banerjee. 1994. Parallel logic synthesis using partitioning. In *1994 International Conference on Parallel Processing*, Vol. 3. IEEE, 135–142.

[21] Kaushik De, L. A. Chandy, Sumit Roy, Steven Parkes, and Prithviraj Banerjee. 1995. Parallel algorithms for logic synthesis using the MIS approach. In *Proceedings of 9th International Parallel Processing Symposium*. IEEE, 579–585.

[22] Shounak Dhar, Love Singhal, Mahesh Iyer, and David Pan. 2019. FPGA accelerated FPGA placement. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. 404–410.

[23] Xiao Dong and Guy G. F. Lemieux. 2009. PGR: Period and glitch reduction via clock skew scheduling, delay padding and GlitchLess. In *2009 International Conference on Field-Programmable Technology*. IEEE, 88–95.

[24] Alfred E. Dunlop, Brian W. Kernighan, et al. 1985. A procedure for placement of standard cell VLSI circuits. *IEEE Transactions on Computer-Aided Design* 4, 1 (1985), 92–98.

[25] John P. Fishburn. 1990. Clock skew optimization. *IEEE Transactions on Computers* 39, 7 (1990), 945–951.

[26] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

[27] Marcel Gort and Jason Anderson. 2014. Design re-use for compile time reduction in FPGA high-level synthesis flows. In *2014 International Conference on Field-Programmable Technology (FPT'14)*. IEEE, 4–11.

[28] Marcel Gort and Jason H. Anderson. 2010. Deterministic multi-core parallel routing for FPGAs. In *2010 International Conference on Field-Programmable Technology*. IEEE, 78–86.

[29] 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.

[30] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[31] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency. In *2020 57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218718

[32] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel physical implementation of FPGA HLS designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–12.

[33] 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 (FPGA'18)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/3174243.3174246

[34] Yutian Huan and André DeHon. 2012. FPGA optimized packet-switched NoC using split and merge primitives. In *2012 International Conference on Field-Programmable Technology*. IEEE, 47–52.

[35] Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 1628–1633.

[36] Abhishek Kumar Jain, Khoa Dang Pham, Jin Cui, Suhaib A. Fahmy, and Douglas L. Maskell. 2014. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems* 77, 1 (2014), 61–76.

[37] Wei Jiang, Zhiru Zhang, Miodrag Potkonjak, and Jason Cong. 2008. Scheduling with integer time budgeting for low-power optimization. In *2008 Asia and South Pacific Design Automation Conference*. IEEE, 22–27.

[38] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[39] Parivallal Kannan and Satish Sivaswamy. 2016. Performance driven routing for modern FPGAs. In *Proceedings of the 35th International Conference on Computer-Aided Design*. 1–6.

[40] Nachiket Kapre and Jan Gray. 2017. Hoplite: A deflection-routed directional Torus NoC for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 2 (2017), 1–24.

[41] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and synthesis for software-defined FPGA acceleration: Status and future prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14, 4 (2021), 1–39.

[42] 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'18)*. IEEE, 133–140.

[43] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. 2011. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 117–124.

[44] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[45] Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan. 2017. UTPlaceF 3.0: A parallelization framework for modern FPGA global placement. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. IEEE, 922–928.

[46] Tao Lin, Chris Chu, and Gang Wu. 2015. POLAR 3.0: An ultrafast global placement engine. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. IEEE, 520–527.

[47] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. 2008. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. 14–23.

[48] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 7, 2 (2014). https://doi.org/10.1145/2617593

[49] Sen Ma, Zeyad Aklah, and David Andrews. 2016. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–178.

[50] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. 2003. Fast timing-driven partitioning-based placement for island style FPGAs. In *Proceedings of the 40th Annual Design Automation Conference*. 598–603.

[51] Pongstorn Maidee, Chris Neely, Alireza Kaviani, and Chris Lavin. 2019. An open-source lightweight timing model for RapidWright. In *2019 International Conference on Field-Programmable Technology (ICFPT'19)*. IEEE, 171–178.

[52] Fubing Mao, Wei Zhang, Bingsheng He, and Siew-Kei Lam. 2017. Dynamic module partitioning for library based placement on heterogeneous FPGAs. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'17)*. IEEE, 1–6.

[53] Michael K. Papamichael and James C. Hoe. 2012. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 37–46.

[54] Dongjoon Park, Yuanlong Xiao, and André DeHon. 2022. Fast and flexible FPGA development using hierarchical partial reconfiguration. In *2022 International Conference on Field-Programmable Technology (ICFPT'22)*. 1–10. https://doi.org/10.1109/ICFPT56656.2022.9974201

[55] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for fast FPGA compilation using partial reconfiguration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL'18)*. IEEE, 235–2353.

[56] Weikang Qiao. 2022. *Customized Computing: Acceleration of Big-Data Applications*. Ph.D. Dissertation. Universoity of California, Los Angeles.

[57] Weikang Qiao, Licheng Guo, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. 2022. TopSort: A high-performance two-phase sorting accelerator optimized on HBM-Based FPGAs. *IEEE Transactions on Emerging Topics in Computing* (2022), 1–15.

[58] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-accelerated near-storage sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. IEEE, 106–114.

[59] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-performance adaptive merge tree sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 282–294.

[60] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. IEEE, 118–125.

[61] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2021. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. *arXiv preprint arXiv:2111.12555* (2021).

[62] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3490422.3502357

[63] Mirjana Stojilović. 2017. Parallel FPGA routing: Survey and challenges. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–8.

[64] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. 2015. Mapping-aware constrained scheduling for LUT-based FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 190–199.

[65] Kizhepatt Vipin, Jan Gray, and Nachiket Kapre. 2017. Enabling partial reconfiguration and low latency routing using segmented FPGA NoCs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–8.

[66] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. 2017. A runtime optimization approach for FPGA routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2017), 1706–1710.

[67] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[68] David Wilson and Greg Stitt. 2019. Seiba: An FPGA overlay-based approach to rapid application development. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig'19)*. IEEE, 1–8.

[69] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. 2020. Fast linking of separately-compiled FPGA blocks without a NoC. In *2020 International Conference on Field-Programmable Technology (ICFPT'20)*. IEEE, 196–205.

[70] Yuanlong Xiao and Andre DeHon. 2022. HiPR: Fast, incremental custom partial reconfiguration for HLS developers. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. Association for Computing Machinery, New York, NY, USA, 155. https://doi.org/10.1145/3490422.3502335

[71] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and AndreĎeHon. 2022. HiPR: High-level partial reconfiguration for fast incremental FPGA compilation. In *2022 32nd International Conference on Field Programmable Logic and Applications (FPL'22)*. IEEE, 1–9.

[72] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. 2022. PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 933–945.

[73] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon. 2019. Reducing FPGA compile time with separate compilation for FPGA building blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT'19)*. IEEE, 153–161.

[74] Xilinx. 2020. Xilinx UltraScale Plus Architecture. (2020). Retrieved May 4, 2023 from https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[75] Xilinx. 2021. (2021). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug905-vivado-hierarchical-design.pdf.

[76] Xilinx. 2021. (2021). https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf.

[77] Xilinx. 2021. (2021). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf.

[78] Zhen Yang, Anthony Vannelli, and Shawki Areibi. 2007. An ILP based hierarchical global routing approach for VLSI ASIC design. *Optimization Letters* 1, 3 (2007), 281–297.

[79] Chao-Yang Yeh and Malgorzata Marek-Sadowska. 2005. Skew-programmable clock design for FPGA and skew-aware placement. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. 33–40.

[80] Michael Xi Yue, Dirk Koch, and Guy G. F. Lemieux. 2015. Rapid overlay builder for Xilinx FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 17–20.

[81] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858.

[82] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A virtualization stack for heterogeneous FPGA clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, 470–483.

[83] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: A multi-layer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 123–134.

[84] Niansong Zhang, Xiang Chen, and Nachiket Kapre. 2020. RapidLayout: Fast hard block placement of FPGA-optimized systolic arrays using evolutionary algorithms. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL'20)*. IEEE, 145–152.

[85] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*. IEEE, 1130–1135.

[86] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. 2015. Area-efficient pipelining for FPGA-targeted high-level synthesis. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.

[87] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and effective placement and routing directed high-level synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–10.

[88] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. 2018. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 269–278.

[89] 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.

[90] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. 2020. Accelerating FPGA routing through algorithmic enhancements and connection-aware parallelization. *ACM Trans. Reconfigurable Technol. Syst.* 13, 4, Article 18 (Aug. 2020), 26 pages. https://doi.org/10.1145/3406959

Received 26 September 2022; revised 15 January 2023; accepted 20 March 2023